Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations 1. Thesis and Dissertation Collection, all items

1992-09

Design of microcontroller software for a satellite-based ferroelectric capacitor experiment

Tutt, Terry Gene

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/23860

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

http://www.nps.edu/library

UNCLASSIFIED

ECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

a REPORT SECURITY CLASSIFICATION
NCLASSIFIED

1b RESTRICTIVE MARKINGS

a SECURITY CLASSIFICATION AUTHORITY

b DECLASSIFICATION / DOWNGRADING SCHEDULE

3 DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution is unlimited

PERFORMING ORGANIZATION REPORT NUMBER(S)

5 MONITORING ORGANIZATION REPORT NUMBER(S)

a NAME OF PERFORMING ORGANIZATION
aval Postgraduate School

6b OFFICE SYMBOL (If applicable)
EC

7a NAME OF MONITORING ORGANIZATION
Naval Postgraduate School

c. ADDRESS (City, State, and ZIP Code)
aval Postgraduate School

7b ADDRESS (City, State, and ZIP Code)
Naval Postgraduate School

a. NAME OF FUNDING / SPONSORING ORGANIZATION

8b OFFICE SYMBOL (If applicable)

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

c. ADDRESS (City, State, and ZIP Code)

10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

1 TITLE (Include Security Classification) DESIGN OF MICROCONTROLLER SOFTWARE FOR A SATELLITE-ASED FERROELECTRIC CAPACITOR EXPERIMENT

2 PERSONAL AUTHOR(S)

3a TYPE OF REPORT
Master's Thesis

13b TIME COVERED
FROM ________ TO ______

14 DATE OF REPORT (Year, Month, Day)
1992 September

15 PAGE COUNT
98

6 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the uthor and do not reflect the official policy or position of the Department of Defense or the US Government.

7 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
microcontroller; satellite-Based Ferroelectric Capacitor; FERRO

9 ABSTRACT (Continue on reverse if necessary and identify by block number) The Naval Postgraduate School's Space ystems Academic Group is developing a satellite-based experiment to evaluate the elecrical properties of ferroelectric capacitors under high levels of ionizing particle adiation. The experiment, called FERRO, is one of three experiments that comprise the PEX mission. The Apex mission is sponsored by the joint Space Test Program and will be aunched in early 1993. The main processor for FERRO is an Intel 80C196KB 16-bit emedded controller that will perform all aspects of experiment control, data acquisition, nd communication with the host satellite processor. The design of systems and communiations software to support the experiment is presented. Functional areas addressed by he design include: microcontroller and peripheral initialization; communication between he experiment and spacecraft processors; fault detection/recovery; recovery from loss f power and development of an IBM PC based program to provide for pre-flight verificaion and testing of experiment hardware and software.

0 DISTRIBUTION AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

2a NAME OF RESPONSIBLE INDIVIDUAL
. PANHOLZER/F. TERMAN

22b TELEPHONE (Include Area Code)
408-646-2948/2178

22c OFFICE SYMBOL
SP EC/Tz

Approved for public release; distribution is unlimited.

Design of Microcontroller Software for a Satellite-Based Ferroelectric Capacitor Experiment

by

Terry G. Tutt
Lieutenant, United States Navy
B.S., Oregon State University, 1987

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

ABSTRACT

The Naval Postgraduate School's Space Systems Academic Group is developing a satellite-based experiment to evaluate the electrical properties of ferroelectric capacitors under high levels of ionizing particle radiation. The experiment, called FERRO, is one of three experiments that comprise the APEX mission. The Apex mission is sponsored by the joint Space Test Program and will be launched in early 1993. The main processor for FERRO is an Intel 80C196KB 16-bit embedded controller that will perform all aspects of experiment control, data acquisition, and communication with the host satellite processor. The design of systems and communications software to support the experiment is presented. Functional areas addressed by the design include: microcontroller and peripheral initialization; communication between the experiment and spacecraft processors; fault detection/recovery; recovery from loss of power; and development of an IBM PC based program to provide for pre-flight verification and testing of experiment hardware and software.

T95834
C.1

TABLE OF CONTENTS

I. INTRODUCTION 1
  A. PURPOSE 1
  B. BACKGROUND 1
    1. APEX Mission 1
    2. Ferroelectric Devices 4
      a. Bistable Polarization 4
      b. Fatigue and Ageing 5
      c. Measuring Polarization 7
    3. The FERRO Experiment 10
      a. Fatigue/Age Testing 11
      b. Ageing Tests 12
      c. Data Collection 13
    4. Intel M80C196KB Microcontroller 14
      a. Timers 16
      b. High Speed Input Unit 17
      c. High Speed Output Unit 18
      d. Serial Port 19
      e. A/D Converter 19
      f. Pulse Width Modulator 20
      g. Watchdog Timer 21
      h. Interrupts 21
    5. FERRO Hardware 23
      a. Memory 24

        b. Communication 24
        c. Programmable Peripheral Interface 24
        d. Ferroelectric Capacitor Input Circuit 25
        e. Ferroelectric Capacitor Output Circuit 26
        f. Temperature Sensors 27
II. SOFTWARE REQUIREMENTS 28
    A. DESIGN GOALS 28
        1. Reliability 28
        2. Ease of Maintenance 29
        3. Flexibility 29
        4. Size and Speed 30
    B. INITIALIZATION AND RECOVERY 30
    C. COMMUNICATION 31
    D. ERROR DETECTION 34
    E. EXPERIMENT CONTROL 34
    F. TEST/VERIFICATION 35
III. SOFTWARE IMPLEMENTATION 36
    A. INITIALIZATION AND RECOVERY 37
    B. COMMUNICATION 38
    C. ERROR DETECTION 42
    D. EXPERIMENT CONTROL 43
        1. FERRO Commands 44
        2. FERRO Experiment Events 48
    E. TEST/VERIFICATION 50
IV. CONCLUSIONS AND RECOMMENDATIONS 52
LIST OF REFERENCES 55

APPENDIX A - FERRO EXPERIMENT HARDWARE SCHEMATIC 56

APPENDIX B - FERRO SOFTWARE SOURCE CODE 60

APPENDIX C - TEST/VERIFICATION SOFTWARE SOURCE CODE 75

INITIAL DISTRIBUTION LIST 89

LIST OF TABLES

TABLE 1. AGEING INTERVALS 13
TABLE 2. 80C196KB INTERRUPT PRIORITIES 22
TABLE 3. FERRO COMMANDS 44

LIST OF FIGURES

Figure 1. APEX Spacecraft and Payload Layout ... 3
Figure 2. Polarization versus Applied Electric Field ... 5
Figure 3. Effect of Fatiguing ... 6
Figure 4. The Sawyer-Tower Circuit ... 7
Figure 5. Current Output for Ferroelectric Capacitor ... 9
Figure 6. FERRO Experiment Cycle ... 11
Figure 7. 80C196KB Block Diagram ... 15
Figure 8. High Speed Input Unit Block Diagram ... 17
Figure 9. High Speed Output Unit Block Diagram ... 18
Figure 10. Analog-to-Digital Converter Block Diagram ... 20
Figure 11. Pulse Width Modulator Block Diagram ... 21
Figure 12. FERRO Experiment Block Diagram ... 23
Figure 13. Ferroelectric Capacitor Input Select Circuit ... 25
Figure 14. Ferroelectric Capacitor Read Circuit ... 26
Figure 15. FERRO Packet Structure ... 31
Figure 16. Receive Interrupt Flow Diagram ... 39

I. INTRODUCTION

A. PURPOSE

The Naval Postgraduate School's Space Systems Academic Group is developing a satellite-based experiment to evaluate the electrical properties of ferroelectric capacitors in a space environment. Although ferroelectric devices have undergone radiation testing in artificial environments, actual testing in space is necessary to assure the suitability of these devices for space applications. The purpose of this thesis is to develop microcontroller software to conduct the experiment, collect data, and provide for communication between the spacecraft and experiment processors.

B. BACKGROUND

1. APEX Mission

The experiment, called FERRO, is scheduled for launch aboard a joint Department of Defense (DOD) spacecraft in March 1993. FERRO is one of three independent experiments that make up the Advanced Photovoltaic and Electronics Experiment (APEX) mission. The mission is sponsored by the joint Space Test Program under the management of the U.S. Air Force.

The objective of the APEX mission is to collect data concerning the effects of a high radiation environment on selected electronic components. To meet this objective, three experiments will be placed in a low-earth, near-polar orbit for a minimum mission life of one year and a goal of three years.

The principal experiment is the Photovoltaic Array Space Power Plus Diagnostics (PASP-PLUS) developed by the NASA/Goddard Space Flight Center. PASP-PLUS will determine the efficiency and performance limits of advanced solar cell designs under stressing space environments. Second, is the Cosmic Ray Upset Experiment (CRUX) developed by the Air Force Geophysics Laboratory. CRUX will develop data to validate and update analytical models used to determine the effects of high energy cosmic rays and trapped protons on space-borne semiconductor components. FERRO is the third and smallest of the experiments and will determine the performance of ferroelectric capacitor test devices exposed to long periods of radiation and high stressed operation.

The APEX spacecraft (Figure 1) is based on the PegaStar integrated spacecraft bus for the Pegasus launch vehicle developed by Orbital Sciences Corporation (OSC). Integration and system level testing of the bus, launch vehicle, and APEX subsystems will take place at OSC's Systems Integration Laboratory at NASA Ames-Dryden Flight Research Facility [Ref. 1].

The APEX spacecraft will be placed in a polar elliptical orbit with a target inclination of 70°, +10°/-0° that will provide periodic exposure to the lower Van Allen radiation belt. Target parameters for initial perigee and apogee were selected to provide a 190 ±10 nmi perigee and a 1054 ±100 nmi apogee over the mission lifetime. [Ref. 1: p. 16]

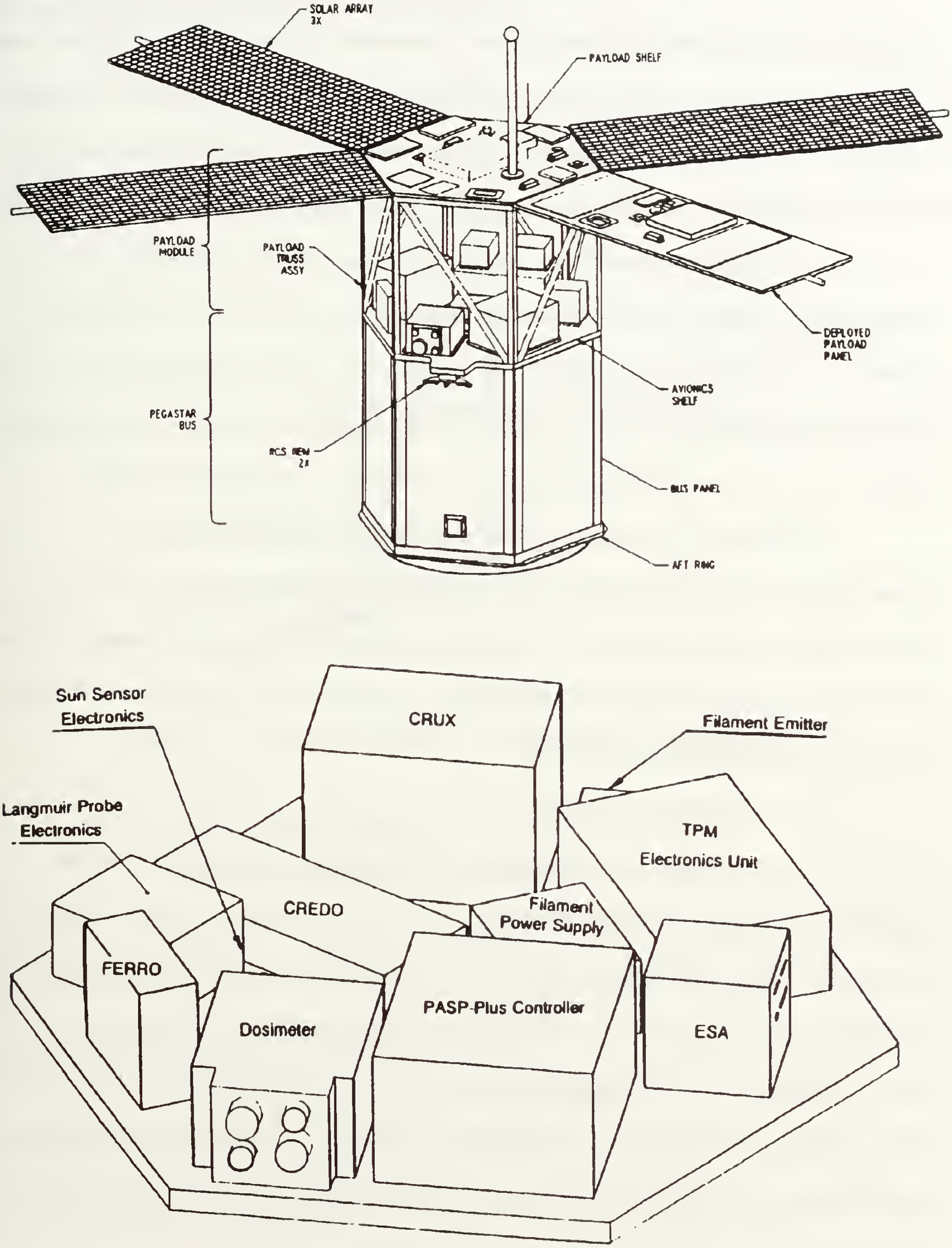


Figure 1. APEX Spacecraft and Payload Layout [Ref. 1]

2. Ferroelectric Devices

A ferroelectric material is one that is polarized by the application of an electric field and that maintains a remanent polarization after removal of the exciting field. Early bulk or thick-film ferroelectric devices required excessive voltages to switch polarization states, but recent advances in thin-film technology have made the use of ferroelectric devices practical.

Thin-film ferroelectric capacitors have been the subject of considerable research over the past decade due to the highly desirable properties these devices exhibit with respect to memory applications. These properties include nonvolatility, radiation-hardness, low power consumption, high speed, high switching endurance, and high component density.

Ferroelectric devices have significant advantages over conventional nonvolatile semiconductor memory technology in many of these areas and this superiority makes ferroelectric memory a natural choice for space-borne and military applications as well as for more conventional commercial applications. [Ref. 2]

a. Bistable Polarization

A basic characteristic of ferroelectric capacitors is the hysteretic behavior relating polarization, P, and the applied electric field, E, as shown in Figure 2 below. [Ref. 3] As the intensity of the electric field is increased, the polarization increases until saturation is reached at P_S. If the electric field is subsequently removed, a remanent polarization, P_r, remains. When the polarity of the electric field is reversed, similar characteristics but of opposite polarity are exhibited. The two zero-field values of remanent polarization, $\pm P_r$, are equally stable and thus result in the ability to provide nonvolatile storage of binary information.

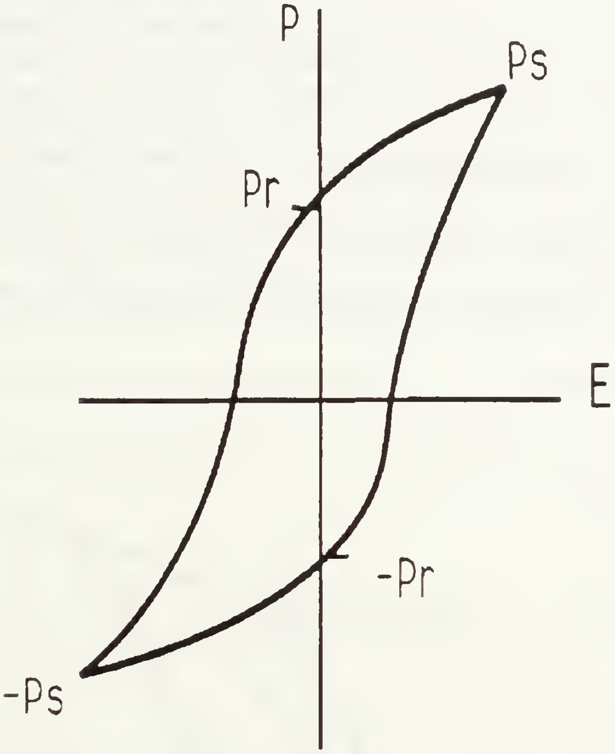


Figure 2. Polarization versus Applied Electric Field [Ref. 4]

b. Fatigue and Ageing

Ferroelectric devices are affected by two primary forms of functional degradation: fatigue and ageing. Standard definitions of these terms have not yet been established, so working definitions for the purpose of this discussion will be described below.

Fatigue is defined as the symmetric loss in remanent polarization that results from repeated application of electric fields [Ref. 4]. The fatiguing effect results in vertical compression of the hysteresis loop as shown in Figure 3. Periodic alternating electric fields are often intentionally applied to ferroelectric materials to establish a measure of the material's "fatigue limit" or the point at which the performance of the device is seriously degraded.

Ageing is defined as the symmetric or asymmetric loss in remanent polarization that occurs under conditions of zero applied electric field following an initial polarization of the material [Ref. 4]. The original polarization levels may sometimes be restored by cycling the device through many polarization reversals by applying an alternating electric field [Ref. 5].

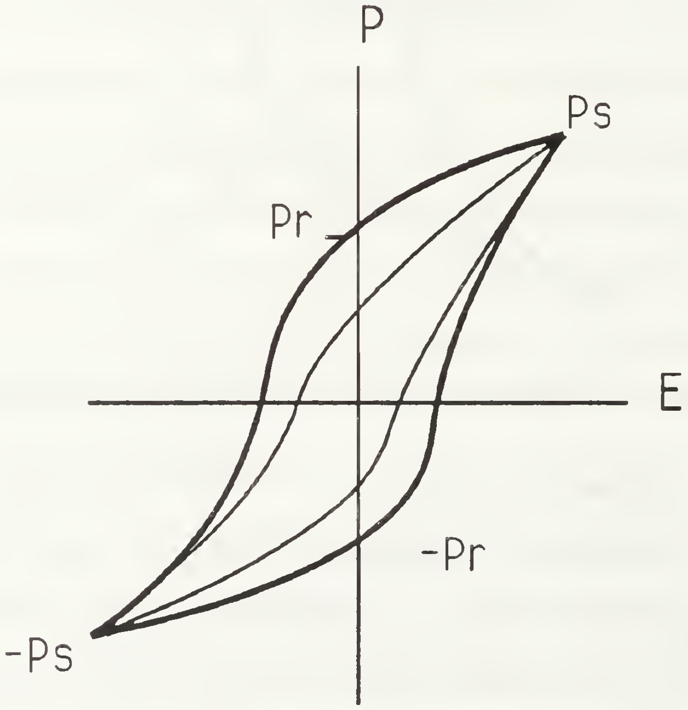


Figure 3. Effect of Fatiguing [Ref. 4]

Experiments to measure the effects of ageing usually consist of polarizing the device and waiting for a specified time before measuring the remanent polarization. Since the effects of fatiguing and ageing are similar, tests to measure the effects of each phenomenon are usually conducted independently.

c. *Measuring Polarization*

Polarization, unlike voltage or current, cannot be measured directly. The Sawyer-Tower circuit shown in Figure 4 provides a simple method for the indirect measurement of polarization of ferroelectric capacitors [Ref. 6: p. 270, Figure 1]. The Sawyer-Tower circuit consists of an integrating or sense capacitor placed in series with the ferroelectric capacitor. The value of the sense capacitor, C_S, is normally chosen to be four to five times the capacitance of the ferroelectric capacitor.

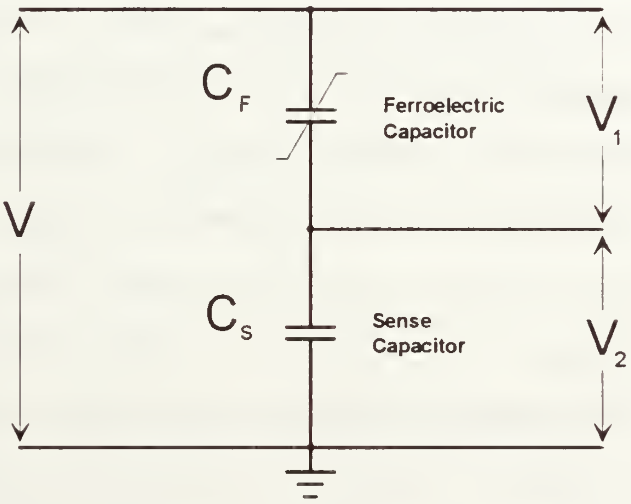


Figure 4. The Sawyer-Tower Circuit [Ref. 4: p. 19]

The charge, q, on the upper plate of the sense capacitor, C_S, (Figure 4) can be calculated from

$$q = C_S V_2 \tag{1}$$

where C_S is the capacitance of the sense capacitor. Since this charge is equal and opposite to the charge on the lower plate of the ferroelectric capacitor, the polarization, P, of the ferroelectric capacitor can be calculated from

$$P = \frac{q}{A} \tag{2}$$

where A is the cross-sectional area of the ferroelectric capacitor. Thus, by combining equations (1) and (2), the polarization of the ferroelectric capacitor can easily be determined by measuring V_2. I[Ref. 4: p. 20]

Unfortunately, connecting measurement equipment to the Sawyer-Tower circuit will affect the charge distribution, thereby making the measurement invalid. This makes it impractical to determine the state of the ferroelectric capacitor statically as described above. Instead, a method involving the application of a "read pulse" to the Sawyer-Tower circuit is used. This method will be explained in detail in the discussion that follows.

Independent of their ferroelectric effects, ferroelectric capacitors have the properties of standard dielectric capacitors. When an electric field is applied to the capacitor, a charging current flows that is proportional to the dielectric constant of the material. If the polarization of the ferroelectric capacitor before applying the electric field is in the same direction as the field, only this charging current will flow.

If the ferroelectric capacitor was polarized opposite the electric field, a larger current flows that is the sum of the charging current of the capacitor and the switching current caused by reversing the polarity of the ferroelectric material. Thus the polarization state of the capacitor before application of the electric field can be determined by the amount of current flow through the capacitor when the field is applied. It should be noted that the ferroelectric capacitor will be polarized in the direction of the applied field after the read operation. Therefore, similar to the operation of dynamic RAM, reading the state of the capacitor destroys the information stored in the capacitor. [Ref. 7: p. 213]

Figure 5 shows the current through a ferroelectric capacitor for the four possible conditions. Note that the capacitor is polarized in the negative direction before the pulse train is applied. When voltage pulse P is applied, a polarization reversal is caused resulting in a larger current pulse. Pulse U causes no polarization reversal and only the smaller charging current flows as illustrated by curve U.

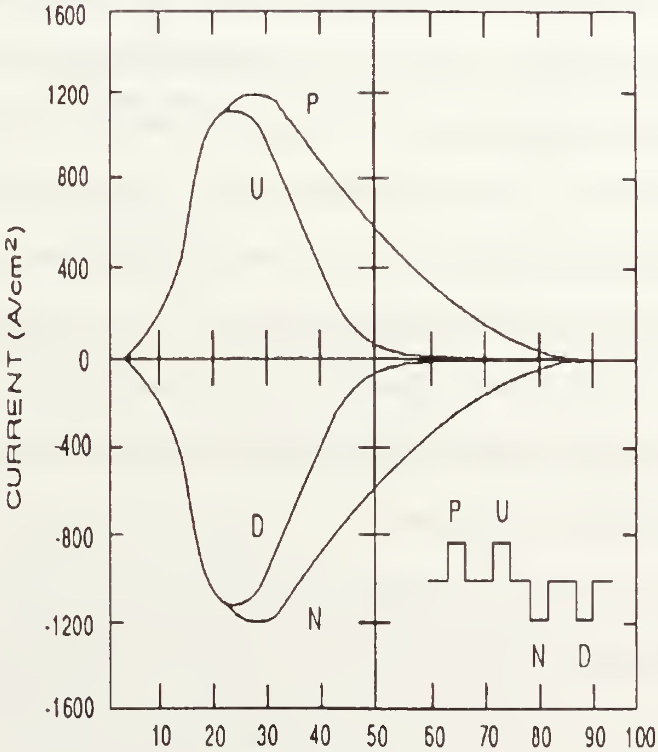


Figure 5. Current Output for Ferroelectric Capacitor [Ref. 7: p. 213]

Similar results occur for the negative pulses as shown by curves N and D. If, due to fatigue or ageing, the ferroelectric capacitor is not fully polarized in a given direction, the current pulse will lie somewhere between the P and U or N and D curves.

When such pulses are applied to a Sawyer-Tower circuit arranged as previously discussed, the sense capacitor will integrate the current pulse and the voltage V_2 (Figure 4) will be proportional to the area under the appropriate curve as shown in Figure 5. Thus by measuring the voltage across the sense capacitor, it is possible to determine the polarization state of the ferroelectric capacitor. A fixed reference direction (positive or negative) for all read pulses is normally chosen.

It should be noted that the Sawyer-Tower circuit is used only for reading the polarization state of the ferroelectric capacitor. The sense capacitor should be switched out of the circuit when an electric field is applied to polarize the ferroelectric capacitor in a given direction. Additionally, the sense capacitor should be in a fully discharged state before being connected to the ferroelectric capacitor. Both requirements are easily met by simply grounding the sense node when polarizing or writing to the ferroelectric capacitor, and removing the ground when reading.

3. The FERRO Experiment

The objective of the FERRO experiment is to measure the fatigue and ageing response of integrated ferroelectric capacitors as they experience high levels of ionizing particle radiation. FERRO contains a total of 32 ferroelectric capacitors divided into four groups of eight referred to as devices under test (DUTs). Two DUTs will serve as the test group. They will be located outside the shielding of the housing subsystem and receive the maximum radiation dose possible. The other two DUTs will serve as the control group and reside inside the shielding of the housing subsystem. Each group is further subdivided so that one DUT will undergo both fatigue and ageing tests, while the other DUT will go through only the ageing tests. The experiment is executed in 21-day cycles as shown in Figure 6. The cycles will be repeated until the end of the experiment.

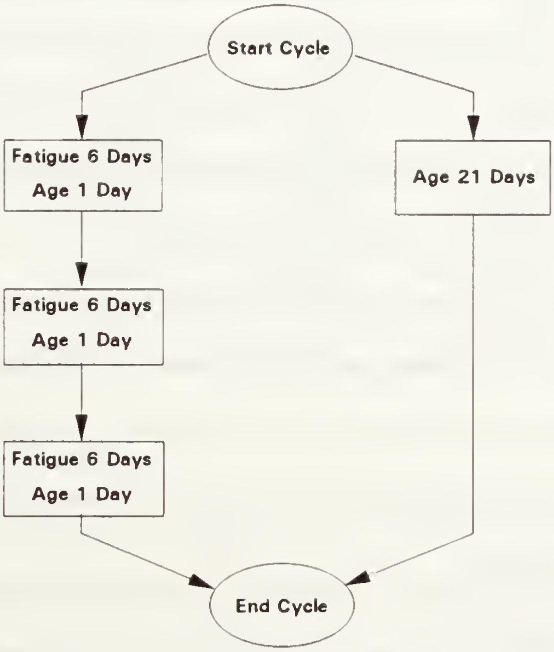


Figure 6. FERRO Experiment Cycle

a. Fatigue/Age Testing

One DUT from each of the control and test groups will undergo a combination of fatigue and age testing. Three identical seven-day subcycles of fatigue/age tests will occur each experiment cycle. During the fatigue phase of testing, a 5 kHz, 10 volt peak to peak bipolar squarewave will be applied to the DUTs with the sense capacitor of the Sawyer-Tower circuit switched out (sense node grounded).

The fatigue phase will continue for six days with a short pause every 24 hours to polarize and then sample the remanent polarization of each ferroelectric capacitor using the Sawyer-Tower circuit. Two data points will be collected for each capacitor. First, a negative pulse is used to write to the ferroelectric capacitor, followed by a read using a positive pulse. This data point will provide a measure of the negative remanent polarization. Using opposite polarities for the read and write pulses ensures that polarization reversal will occur during the read process. For the second data point, the polarity of the read and write pulses are reversed to provide a measure of the positive remanent polarization.

On the seventh day of each fatigue/age subcycle, the DUTs will undergo ageing tests that are identical to the ageing tests for the ageing DUTs described below, but limited to 24 hours in duration. The 24-hour period of ageing tests will allow for tests using the first 23 time intervals of Table 1 to be completed.

b. Ageing Tests

The other DUT in each of the control and test groups will undergo only ageing tests for the 21-day experiment cycle. Each ageing test will consist of writing to the ferroelectric capacitors and waiting a specified ageing time before reading the capacitors. The read and write pulses will be of opposite polarity, as in the fatigue testing, and the polarities will be reversed each experiment cycle. The ageing intervals for the 30 ageing tests done each experiment cycle are shown in table 1.

The cumulative time to complete all 30 ageing tests is approximately 20 days, 7 hours, but the ageing test cycle was rounded to 21 days to keep the fatigue/ageing and ageing test cycles in synchronization. This also allows the test events to occur at fixed times in relation to the 24-hour clock.

TABLE 1. AGEING INTERVALS (seconds)

1	2	3	4	5
10	20	30	40	50
100	200	300	400	500
1,000	2,000	3,000	4,000	5,000
10,000	20,000	30,000	40,000	50,000
100,000	200,000	300,000	400,000	500,000

c. Data Collection

Each fatigue or ageing measurement will be an analog voltage that must be converted and stored in binary format. In addition to the ferroelectric capacitor measurements, internal and external temperature will be measured and recorded regularly. This data, along with other experiment status information will be uploaded to the APEX processor periodically. The APEX processor will collect data from all three experiments and relay the data to earth. Radiation dosimetry data will be provided along with the FERRO experiment data.

4. Intel M80C196KB Microcontroller

The M80C196KB 16-bit microcontroller was chosen as the processor for the FERRO experiment. Details of the selection process are discussed in Reference 8. The M80C196KB is a highly integrated single chip device that incorporates many commonly used peripherals. The processor is a member of the CHMOS branch of the MCS-96 family and shares a common architecture and instruction set with other members in the family such as the 8096. The CHMOS devices have enhancements to provide higher performance at lower power consumption than the other devices in the family.

A radiation-hard version of the processor is not available. While there is no data concerning radiation tolerance or single-event-upset rate for the chip, the fabrication process used by Intel for their military products has been shown to exhibit high total dose thresholds [Ref. 8: p. 59].

A simplified block diagram of the M80C196KB is shown in Figure 7. The CPU has a register-register architecture and most operations can be quickly performed from or to any register. A 256-byte register space includes a 232-byte general-purpose register file that can be accessed as bytes, words (16 bits), or double-words (32 bits). Control of on-chip peripherals is accomplished through Special Function Registers (SFRs), which make up the balance of the register space.

On-chip peripherals include a full duplex serial port, an A/D converter, a pulse-width-modulator,up to 48 I/O lines (depending upon which peripherals are used), a high-speed I/O subsystem, two timers, and a watchdog timer.

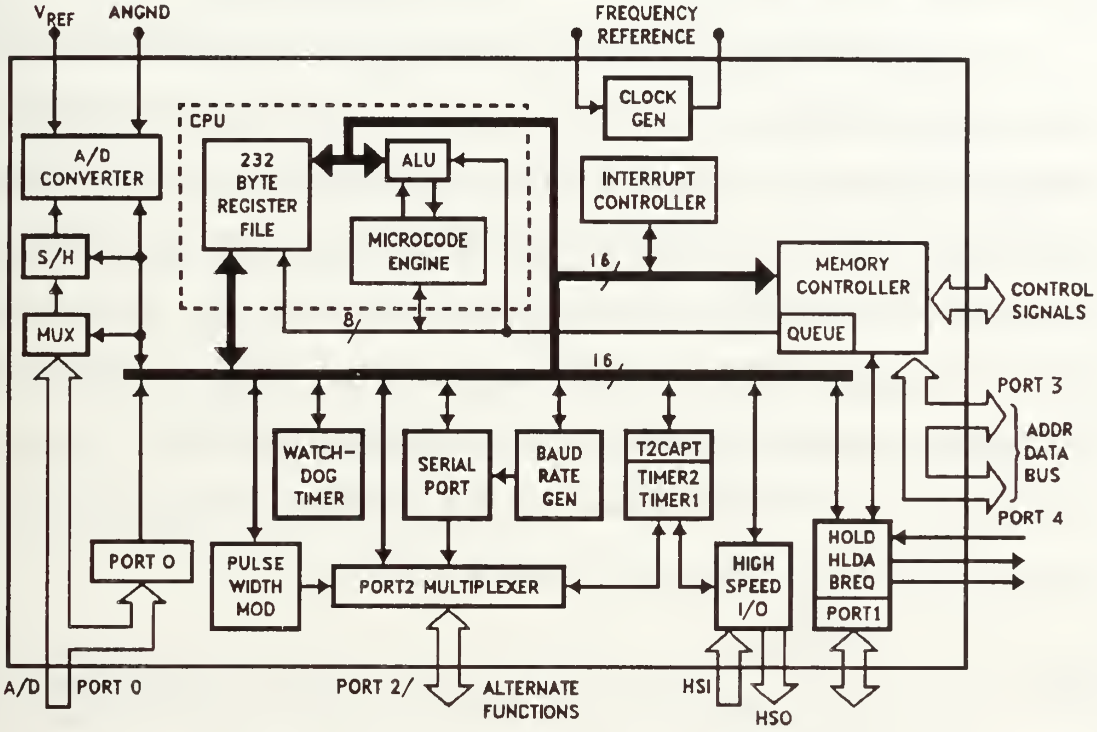


Figure 7. 80C196KB Block Diagram [Ref. 9: p. 4-1]

The 80C196KB has an addressable memory space of 64 Kbytes. The 256 bytes from 0000H through 00FFH are used for the SFRs and register file, while the locations between 2000H and 2080H are reserved for interrupt vectors and a chip configuration byte. A pin that is asserted on instruction fetches is provided to allow decoding of more than 64 Kbytes of addressing space. If used, this signal will allow for 64 Kbytes of code space and 64 Kbytes of data space.

The 80C196KB can be runtime configured to operate with either a 16-bit or 8-bit data bus. For 16-bit bus cycles, Ports 3 and 4 contain the address multiplexed with data. For 8-bit bus cycles, Port 3 is multiplexed with address and data, while Port 4 contains only the most significant byte of address. The bus width can be changed every bus cycle. Several modes of bus control are available to match a wide variety of external peripherals. This versatility reduces the external interface circuitry required. Additionally, the 80C196KB supports a bus exchange protocol to allow other devices to gain and return control of the bus.

a. Timers

Two 16-bit timers are available on the 80C196KB. Timer1 is a free-running timer that is incremented every eight state times (sixteen system clock periods). Timer1 may be used as a reference for both the HSI and HSO units. An interrupt can be generated when the timer overflows.

Timer2 counts both positive and negative transitions on its selected input. It can be configured as either an up counter or down counter. Timer2 can be reset by hardware, software, or the HSO unit. Either Timer1 or Timer2 may be used as a reference for the HSO unit. The maximum transition speed is once per state time in Fast Increment mode and once every eight states otherwise. The value of Timer2 may be captured into a designated register by a rising edge on an external pin. An interrupt can be generated by the capture operation. Interrupts can also be generated by crossing either the 0FFFFH/0000H boundary or the 7FFFH/8000H boundary in either direction.

b. High Speed Input Unit

The High Speed Input (HSI) unit, shown in Figure 8, can capture the Timer1 value when an event occurs on one of four input pins. Four types of events may be specified: rising edge, falling edge, rising or falling edge, or every eighth rising edge. A FIFO and holding register allow up to eight events to be recorded before they must be read. The HSI unit can generate interrupts in three ways: when a value moves into the holding register (i.e. data is available), when four events have been recorded, or when six events have been recorded.

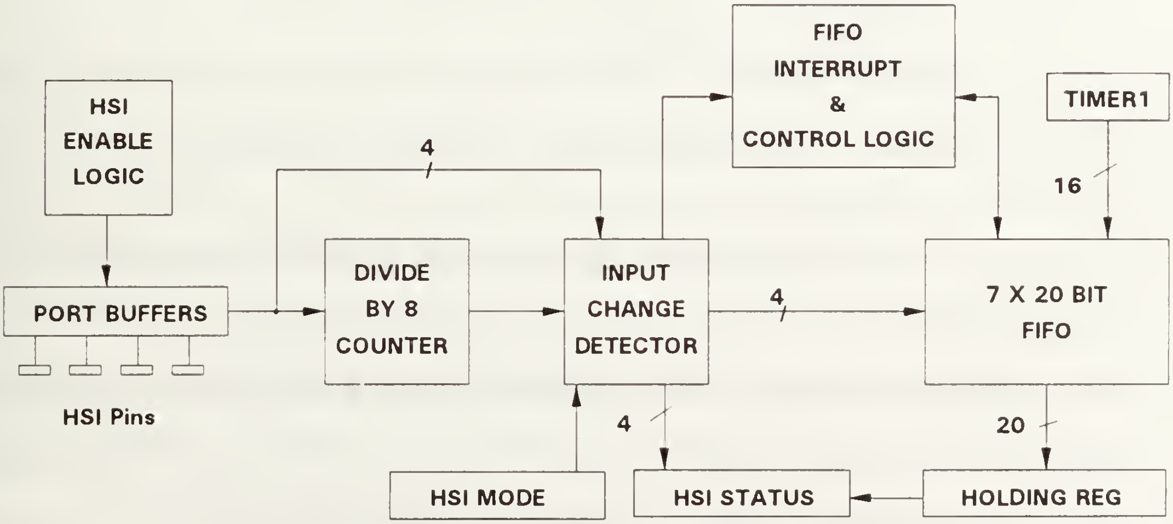


Figure 8. High Speed Input Unit Block Diagram [Ref. 9]

c. High Speed Output Unit

The High Speed Output (HSO) unit can schedule and generate events at specified times based on Timer1 or Timer2. A block diagram of the HSO is shown in Figure 9. These events can be scheduled to occur once, or to be recurrent, with minimal CPU overhead. Up to eight pending events can be scheduled and stored in a Content Addressable Memory (CAM). One CAM register is compared with the timer values each state time.

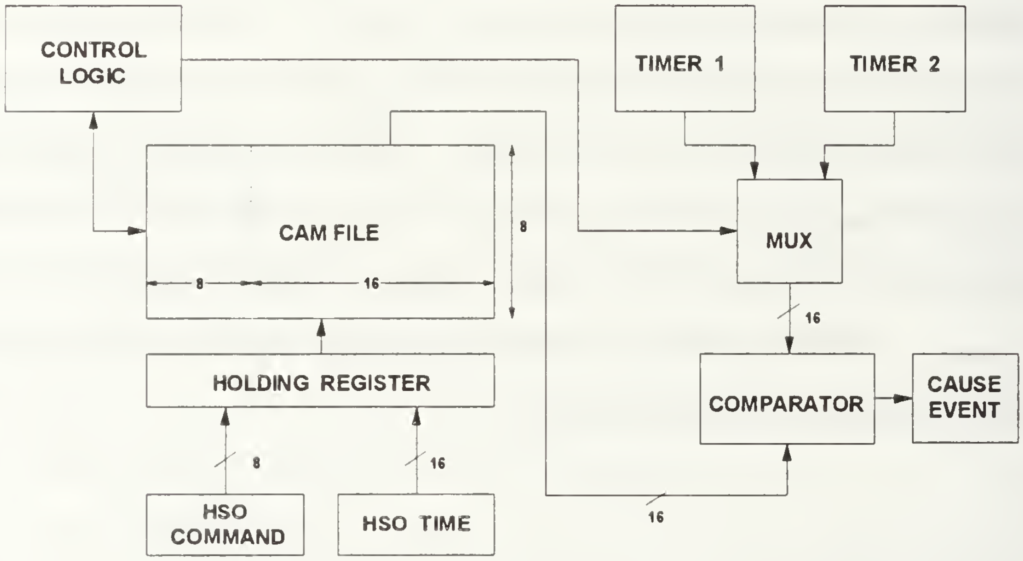


Figure 9. High Speed Output Unit Block Diagram [Ref. 9]

One CAM register is checked each state time to determine if an event should be initiated. The minimum time resolution of the HSO is eight state times since all eight CAM registers must be checked. Events that may be scheduled include: starting an A/D conversion, resetting Timer2, setting four software timers, and switching six output lines. Interrupts can be generated by any of these events.

d. Serial Port

The 80C196KB provides an onboard serial port with one synchronous mode and three full duplex asynchronous modes. Two of the asynchronous modes are designed for interprocessor communications. Double buffering is provided for both the transmitter and receiver. Baud rates are generated with an independent 15-bit counter based on either the system clock or the Timer2 clock.

The serial port may generate one of three maskable interrupts: Transmit Interrupt (TI), Receive Interrupt (RI), or SERIAL. A SERIAL interrupt is generated on both transmit and receive cycles, while TI and RI are generated on transmit and receive respectively.

e. A/D Converter

The 80C196KB's built-in A/D converter is shown in Figure 10. It consists of an 8-channel multiplexer, a sample-and-hold circuit, and a 10-bit successive approximation analog-to-digital converter.

Any one of the eight analog input pins (shared with Port 0) can be sampled under the control of the A/D Command register. Additionally, the conversion process can be scheduled using the High Speed Output (HSO) unit. The result is a 10-bit binary representation of the ratio of the analog input voltage divided by the analog reference voltage, V_{REF}.

The conversion process can take either 91 or 158 state times depending upon whether a prescaler bit is used. The prescaler is necessary if higher clock speeds are used, to allow time for the comparator to settle during the conversion. An interrupt can be generated upon completion of the conversion process.

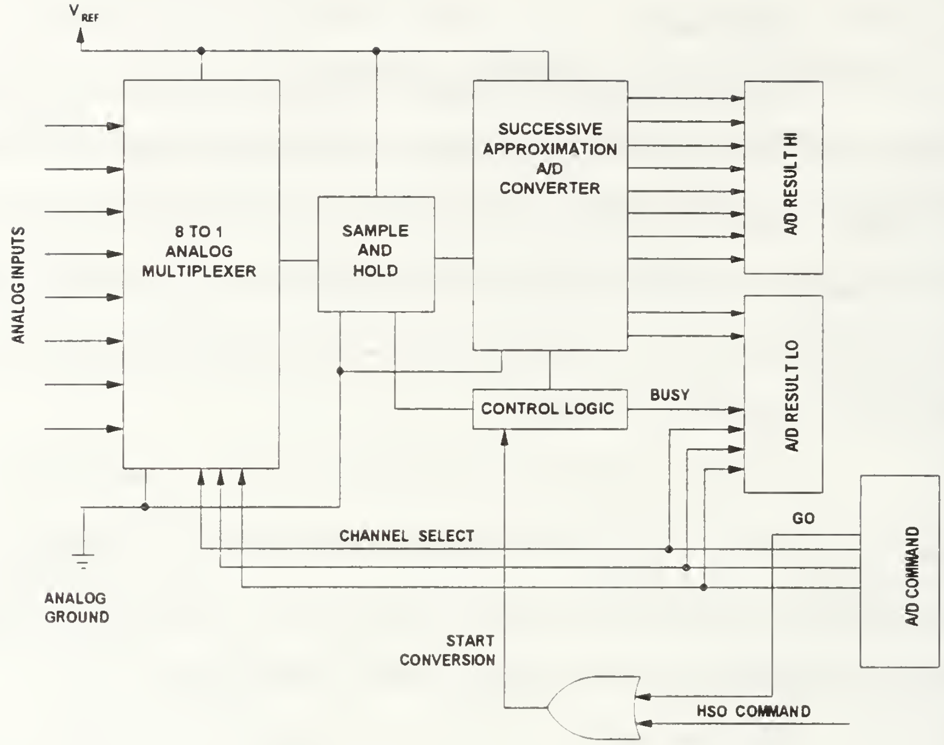


Figure 10. Analog-to-Digital Converter Block Diagram [Ref. 9]

f. Pulse Width Modulator

The Pulse Width Modulator (PWM) generates a variable duty cycle pulse that repeats every 256 or 512 state times depending upon whether a pre-scaler (divide by 2) is enabled. A block diagram of the PWM circuit is shown in Figure 11. An 8-bit counter is incremented every state time. When the counter overflows, the output is set high, when it matches the PWM register, the output is set low. The duty cycle of the output waveform is determined by the value in the PWM register. Applications for the PWM include generating motor drive control signals and performing digital-to-analog conversion.

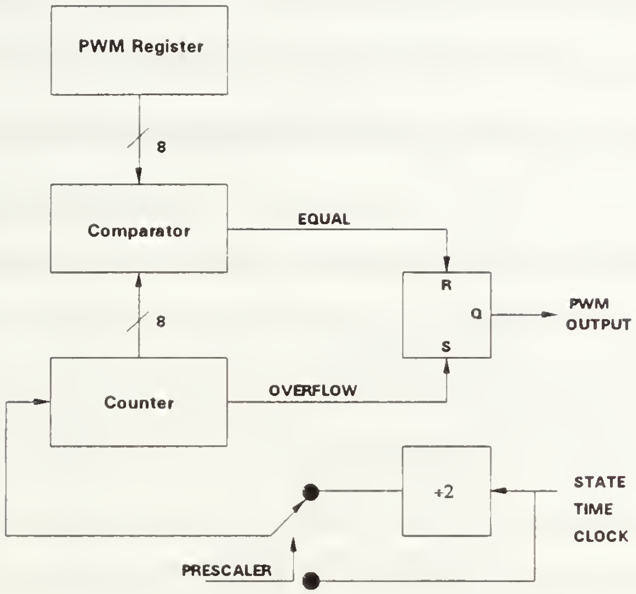


Figure 11. Pulse Width Modulator Block Diagram [Ref. 9]

g. Watchdog Timer

The Watchdog Timer allows for graceful recovery from a software fault. When enabled, the watchdog will initiate a hardware reset unless the timer is cleared by software before it overflows. The 16-bit timer is incremented every state time; thus the programmer must ensure that under normal operation, the watchdog timer is cleared at intervals not to exceed 64K state times.

h. Interrupts

The 80C196KB microprocessor has 28 sources of interrupts mapped onto 18 vectors. Table 2 shows the interrupts and their priorities with 15 being the highest priority and 0 the lowest. The interrupts are individually maskable with the exception of the Nonmaskable interrupt, the Unimplemented Opcode interrupt and the Trap interrupt. Interrupts may be globally enabled or disabled by changing a flag in the Program Status Word. Several instructions are provided to control the state of this flag.

TABLE 2. 80C196KB INTERRUPT PRIORITIES [Ref. 8]

Interrupt Number	Source	Priority
INT 15	NMI	15
INT 14	HSI FIFO Full	14
INT 13	EXTINT1	13
INT 12	TIMER2 Overflow	12
INT 11	TIMER2 Capture	11
INT 10	4th Entry into HSI FIFO	10
INT 09	RI	9
INT 08	TI	8
SPECIAL	Unimplemented Opcode	N/A
SPECIAL	Trap	N/A
INT 07	EXTINT	7
INT 06	Serial Port	6
INT 05	Software Timer	5
INT 04	HSI Pin 0	4
INT 03	HSO Output Pins	3
INT 02	HSI Data Available	2
INT 01	A/D Conversion Complete	1
INT 00	Timer Overflow	0

When an interrupt is detected, the corresponding bit is set in one of two interrupt pending registers. Interrupts are processed after the instruction that is currently executing is completed (instructions are indivisible). If the interrupt signal does not occur prior to four state times before the end of the instruction, the interrupt will not be processed until after the following instruction. The maximum interrupt latency, based on the longest instruction, is 61 state times. When the interrupt vector is taken, the appropriate bit is cleared in the interrupt pending register.

5. FERRO Hardware

The basic design for the FERRO experiment hardware is presented in Reference 8. Several changes have been made to the initial design and the updated schematic diagram is included as Appendix A. The major components of FERRO include the 80C196KB microcontroller, a Programmable Peripheral Interface, static Random Access Memory (RAM), Read Only Memory (ROM), temperature sensors, analog drivers, analog switches and multiplexers, and the ferroelectric capacitors (DUTs). Figure 12 is a block diagram that shows the relationship between the major components.

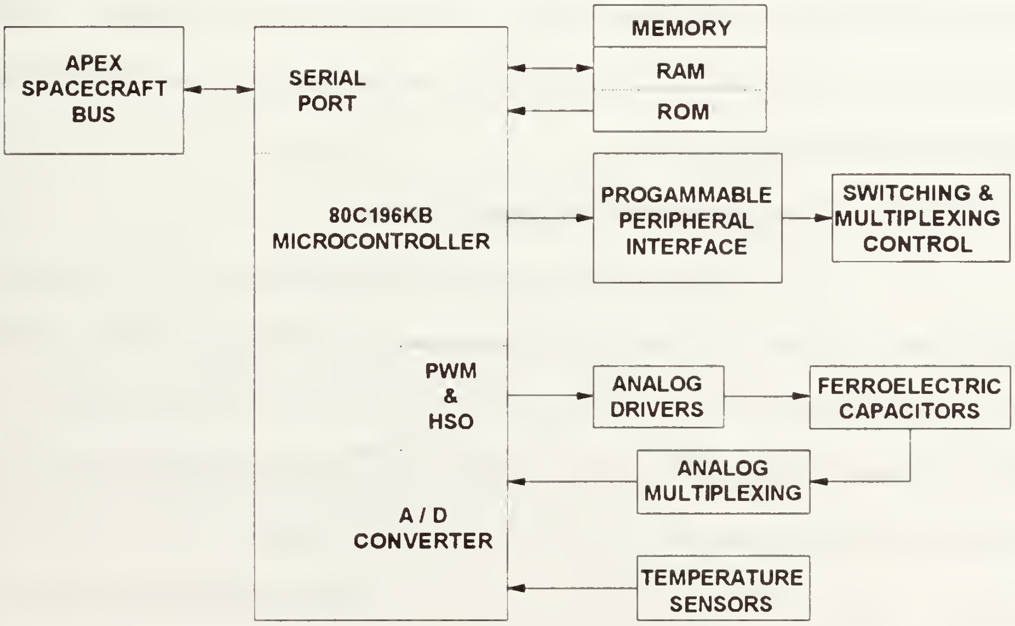


Figure 12. FERRO Experiment Block Diagram

a. Memory

The microcontroller memory consists of 32 Kbytes of ROM and 8 Kbytes of static RAM. The ROM is used for program storage, while the RAM is used as a scratchpad and for temporary storage of status and experimental data. The ROM is mapped to the lower half of the memory space. A radiation-tolerant 32-Kbyte ROM chip was chosen for the experiment. The system RAM consists of a radiation hard silicon-on-sapphire 8-Kbyte static RAM chip. The chip select logic was modified from the original design to provide for more flexibility in adding memory devices. Up to four 8 Kbyte memory devices are allowed for above program memory.

b. Communication

The FERRO experiment will communicate with the APEX spacecraft processor using the 80C196KB serial port. A packet switching protocol will be used for APEX to send commands to FERRO and for FERRO to send status and experiment data to the spacecraft processor. The EIA RS-422 two-way differential hardware interface is used with no hardware handshake lines implemented.

c. Programmable Peripheral Interface

The Programmable Peripheral Interface (PPI) is used to extend the number of output pins available to control the analog switching and multiplexing circuitry. A radiation-hard version of the 82C55A is used which provides three 8-bit ports. The original design called for the input to the PPI to be provided from Port 1 on the 80C196KB. This would have required interface control signals to be generated by the microcontroller under software control. A memory-mapped interface to the PPI was chosen instead. The memory-mapped interface requires memory decode logic to generate the chip select, but has no software overhead.

d. Ferroelectric Capacitor Input Circuit

The PWM and HSO circuits on the microcontroller generate pulses to fatigue, read, and write the ferroelectric capacitors. A block diagram illustrating the select circuitry for one of the four DUTs is shown below in Figure 13. Analog drivers provide isolation and provide for both positive and negative pulses. The PWM generates a continuous train of pulses for fatiguing. Switches are provided to either apply the fatigue pulses to the ferroelectric capacitors or to ground the input of the capacitors. Read and write pulses are generated by the HSO. Analog switches select the output of either noninverting or inverting amplifiers to apply positive or negative pulses. Fatigue pulses are applied to DUTs 1 and 2 only, while read and write pulses are applied to all four DUTs.

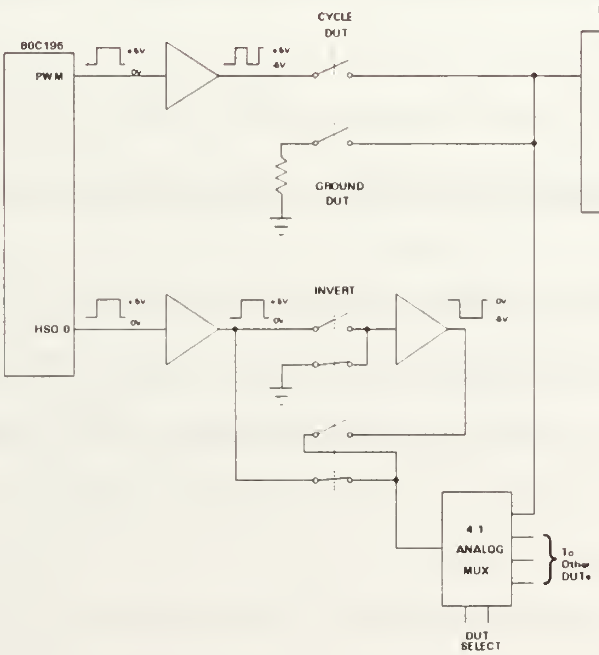


Figure 13. Ferroelectric Capacitor Input Select Circuit

e. Ferroelectric Capacitor Output Circuit

The analog voltage output from the individual Sawyer-Tower circuits associated with each ferroelectric capacitor is applied through a series of analog multiplexers to the analog-to-digital converter on the 80C196KB. Figure 14 below shows a block diagram of the output circuit for a typical DUT.

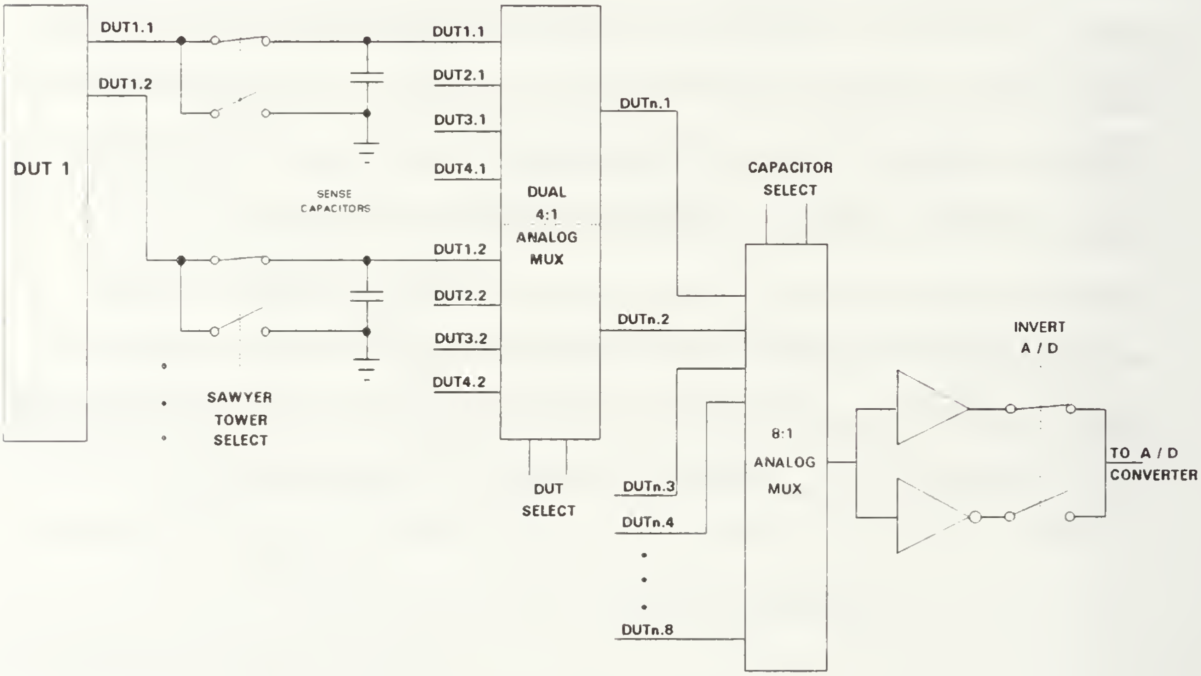


Figure 14. Ferroelectric Capacitor Read Circuit

When enabled during a read operation, the Sawyer-Tower Select switches will remove the ground from the bottom of the ferroelectric capacitors and switch in the sense capacitors. When not enabled, the switches ground the bottom of the ferroelectric capacitors. In addition, the sense capacitors are removed from the circuit and discharged. The eight Sawyer-Tower circuits for each DUT are operated as a unit.

The outputs of the Sawyer-Tower circuits are applied to eight 4:1 analog multiplexers that select output signals from the proper DUT. An 8:1 analog multiplexer then selects the desired channel from the eight available inputs. It should be noted that only one channel of the analog-to-digital converter is needed to measure any of the 32 ferroelectric capacitors.

The analog-to-digital converter on the 80C196KB can convert only positive input voltages. Since the output from the selected Sawyer-Tower circuit will be negative for negative read pulses, an inverting operational amplifier is switched in for these conversions. A noninverting buffer amplifier is selected for positive voltages.

f. Temperature Sensors

Four temperature sensing circuits are provided to monitor the ambient temperature of the DUTs during the experiment. Two thermistors are mounted adjacent to the external DUTs and two are mounted adjacent to the internal DUTs. Changes in the thermistor's resistance are sensed by an operational amplifier circuit that produces an output voltage proportional to the temperature of the thermistor. The circuits sense temperatures over a range of -40°C to +70°C with an accuracy of ±1°C. The output of the sensing circuit is applied through a buffer amplifier to the analog-to-digital converter on the 80C196KB. Each of the four sensing circuits have a dedicated channel on the analog-to-digital converter.

II. SOFTWARE REQUIREMENTS

A. DESIGN GOALS

The Intel 80C196KB microcontroller will perform all aspects of experiment control, data acquisition, and communication with the host satellite processor. The design of the FERRO system and experiment software should not only meet the specifications of the experiment; it should maximize the probability of the successful conduct of the experiment given the hardware design. To meet this overall objective, the following design goals for the FERRO software have been established in order of precedence: reliability, ease of maintenance, flexibility, small size, and speed.

1. Reliability

Since the experiment will be conducted in space, reliability of the software is of primary importance. It should be thoroughly tested and error-free before the experiment is delivered for spacecraft integration. Critical regions where interrupts could result in nondeterministic program behavior should be identified and accounted for. A conservative approach should be taken with respect to hardware timing constraints wherever possible.

In addition to the reliability of the software itself, the reliability of the hardware components should be considered in the software design. Since the APEX spacecraft will be in a high-radiation orbit, the possibility of single-event-upsets (SEU) and hard faults must be considered and allowed for within the constraints of the selected hardware. The design of the software should provide for fault detection and recovery and should allow for redundancy where possible. Cost and availability have limited the use of radiation-hard components, so the radiation-hard components that were chosen should be used for permanent storage or backup of data.

2. Ease of Maintenance

Both during the development cycle and after completion, it is important that the software be easy to maintain. The software design should allow for changes to be integrated efficiently and without introducing error. The software should be easily understood by follow-on programmers and engineers.

To meet this goal, the design of the software should be kept simple and straightforward. Modular design and structured programming concepts should be used extensively. A high-level language should be used wherever possible. Documentation of the program should be thorough, clear, and concise. Constants and variables with descriptive names should be used to make the code easier to read and understand. Future expansion and changes in specifications and hardware should be anticipated and allowed for in the design where practical.

3. Flexibility

The software should give the ground crew as much flexibility as possible in controlling the experiment. The more options available to the ground crew, the more likely they will be able to adjust to the demands of unforeseen events and changes that may occur during the conduct of the experiment. The extent of the flexibility provided may ultimately be limited by the hardware design, but this concept should be kept in mind throughout the design of the software.

4. Size and Speed

Size and speed are not overriding considerations in the software design for this experiment, but these attributes will be optimized when doing so does not compromise the other design goals. While this may not provide direct advantages, keeping the code small and fast may give rise to side benefits such as allowing for redundant code or a more flexible command structure.

B. INITIALIZATION AND RECOVERY

The FERRO software is responsible for initializing and configuring the 80C196KB microcontroller and the onboard peripherals when the experiment is powered on. Externally, the 8255 Programmable Peripheral Interface must be initialized and the analog switches and multiplexers must be set to apply ground to the ferroelectric capacitors. Once initialized the program will stand by for commands from the spacecraft processor to start the experiment.

Since the FERRO experiment is considered a noncritical load on the APEX spacecraft, it is possible that the experiment may be shut down for periods to conserve power. In this event, orderly shutdown and recovery procedures must be provided to allow the experiment to continue when power can be restored. Experiment state information and any data that has been collected should be sent to the spacecraft processor.

In the event of an unforeseen loss of power or in the case of a soft fault causing the watchdog timer to time out, the software should recover gracefully and wait for further commands.

C. COMMUNICATION

The FERRO experiment will communicate with the APEX spacecraft processor using the OX.25 packet-switching protocol. The data will be transmitted as asynchronous bytes at a data rate of 9600 bits per second with one start bit, one stop bit, eight data bits and no parity. OX.25 is based on the Motorola X.25 protocol and is fully described in Reference 10. Figure 15. below shows the packet structure.

FLAG	SIZE	DESTINATION	SOURCE	CONTROL	DATA	CHECKSUM

Figure 15. FERRO Packet Structure

The flag is a single byte, 07EH, that signifies the start of the packet. If a flag character occurs as data within the packet, an extra flag character will be inserted into the data stream as the packet is transmitted to serve as an escape sequence. When the packet is received, the extra flag character must be removed. Any single flag received in the packet following the first byte is an error and the packet must be ignored.

Multi-byte integers can be stored by computer systems in two ways. The values may be stored with the most significant byte (MSB) first (at a lower address) with the remaining bytes following in order of significance. This is referred to as big endian format. Another format, called little endian, stores the bytes in reverse order with the least significant byte first. In both cases the address of the first byte is used to refer to the value.

Microprocessors normally use one of these formats. Values must be stored in the specified format to be operated on correctly by the processor. Values can be converted between the two formats by simply reversing the order of the bytes.

The two-byte size field specifies the length of the data field. The field is stored using big endian format. The source and destination fields are two-byte codes specifying the source and destination addresses of the packet. If FERRO receives a packet with an address that does not match the FERRO address, the packet must be ignored. All packets sent by FERRO will have the APEX spacecraft address specified in the destination field.

Only two bits of the two-byte control field are implemented for this experiment. The ACK request bit (bit 3) is set to request an acknowledgment packet in response to the command. The ACK response bit (bit 2) is set in the packet sent in response. All commands sent by the spacecraft processor will have the ACK request bit set and thus will require a response.

The two-byte checksum field is used to detect errors that could occur during transmission. The checksum encoding and decoding algorithm is specified in Reference 10 and is presented below.

Assume:

Len is the length of the packet.

P is an array of length Len with elements numbered from 0..Len-1.

C0 and C1 are one-byte checksum accumulators.

Encoding Algorithm:

```
Initialize:
        P[Len-1] = P[Len-2] = 0;
        C0 = C1 = 0;
Compute:
        for i := 0 to Len -1 do
           C0 := C0 + P[i];
           C1 := C0 + C1;
Store results:
        P[Len-2] := C0 - C1;
        P[Len-1] := C1 - 2×C0
```

Decoding Algorithm:

```
Compute:
        for i := 0 to Len -1 do
           C0 := C0 + P[i];
           C1 := C0 + C1;
Result:
        Checksum is valid if both
        C0 and C1 are zero.
```

If an invalid checksum is received, the packet will be ignored.

The spacecraft processor will initiate all communication in the form of commands to the FERRO processor. Commands will all have a four-byte data field with the first byte containing a code specifying the command. The remaining three bytes can contain supplementary data with any unused bytes set to zero. FERRO will respond either with a packet containing the requested data or with an acknowledgment packet.

If the spacecraft processor does not receive a valid packet in response to a command, the command will be reissued. Transmission of the response packet must be completed within two seconds after FERRO receives the command or the spacecraft processor will assume the command will not be acknowledged.

D. ERROR DETECTION

The FERRO experiment will be required to operate in a harsh environment over a long period. Use of the watchdog timer for detecting catastrophic errors will prevent the microcontroller from being locked in an infinite loop. In addition, methods for detecting more subtle errors are necessary to ensure the integrity of the data gathered. Both the program firmware and the experiment data buffers should be checked for errors periodically. In the case of a soft fault, recovery can be initiated and the experiment can continue. In the case of a hard fault, recovery may not be possible.

E. EXPERIMENT CONTROL

The experiment will be conducted under software control as modified by commands issued either from the ground or automatically by the spacecraft processor. The software will initiate and control the fatigue and ageing cycles of the experiment as defined in Table 1 and Figure 6. Ferroelectric capacitor, temperature and status data will be collected at appropriate times and sent to the spacecraft processor when commanded. The spacecraft processor will temporarily store the data and periodically downlink the data from all three experiments to the ground station.

A real-time experiment clock will be maintained to act as a reference for experiment events and to time-tag data. The experiment clock will be referenced to the spacecraft universal time and will be synchronized to universal time periodically.

F. TEST/VERIFICATION

A system to simulate the spacecraft processor will be required for testing during software development and for verifying the functional operation of the experiment at delivery. The system must be able to transmit all commands that can be issued by the spacecraft. It must also receive the responses from the experiment hardware. The responses will be displayed and evaluated for accuracy. The system should allow a permanent log of the testing to be saved for later analysis.

III. SOFTWARE IMPLEMENTATION

The Intel iC-96 C compiler was chosen for development of the FERRO software. The compiler provides for easy and direct manipulation of the microcontroller's SFRs while maintaining the advantages of a high level language. A number of pragmas, or directives to the compiler, are available to give the programmer precise control over the resources offered by the 80C196KB microcontroller. The compiler also supports in-line assembly language for use where speed is essential. Reference 11 gives a complete description of the compiler and relocatable linker.

An IBM PC based development system with an 80C196 emulator was used for testing and debugging during software development. The emulator is discussed in Reference 12. A pod that plugged into the processor socket on the experiment prototype board allowed the emulator to interface with peripherals as they were added to the board. The system provides for emulation of both ROM and RAM. Features of the system include software breakpoints, single and multiple step operation, and easy access to SFRs and memory.

Almost all variables are stored in RAM in order to take advantage of the radiation-hard component used. Only variables used in operations where time is a consideration are stored in registers. A compiler pragma is used to disable the storage of variables in registers except when explicitly declared. The source code for the FERRO experiment is included as Appendix B. The source was divided into several files by functional area for organizational purposes and to allow for changes to the code to be made without recompiling the entire program. A make utility was used to aid in the compiling and linking of the source files.

A. INITIALIZATION AND RECOVERY

Initialization routines for the microcontroller and programmable peripheral interface are called immediately after the experiment is powered up. The 8255 is set to provide three 8-bit output ports for analog switching and multiplexer control. Initial values are sent to the ports to ground the ferroelectric capacitors.

A compiler pragma statement sets the Chip Configuration Byte (CCB). The CCB is loaded into the Chip Configuration Register whenever the 80C196KB is reset and on power up. For FERRO, the CCB is set for an 8-bit bus width and to use the Address Valid Strobe bus control signals. These settings were chosen to be compatible with peripherals used.

The serial port is initialized to provide asynchronous communications at 9600 bits per second. The baud rate is based on the system clock of 4.9152 MHz. The value for the baud register can be calculated from the formula shown in equation (3).

$$\text{Baud Register} = \frac{\text{System Clock}}{\text{Baud Rate} \times 16} - 1 \tag{3}$$

The most significant bit of the 16-bit baud register must be set to select the system clock as the reference. Substituting values into equation (3) and setting the MSB yields a value of 801FH. The value must be loaded into the baud register as two sequential bytes, with the low byte loaded first. This method is obscurely described in one sentence of Reference 9 and was discovered only after several hours of frustration. Once the serial port is initialized, the receive (RI) interrupt is unmasked and interrupts are enabled.

The HSO Software Timer 0 is set to initiate an interrupt every 0.125 seconds. The interrupts are used to update the real-time clock for the experiment. Timer 2 is the reference for the HSO commands that implement the real-time clock and an HSO command is used to reset the timer each time the interrupt is generated. These commands are all locked in the HSO CAM to produce the desired periodic interrupts without further processor overhead.

The Timer 2 Clock input is derived from the CLKOUT pin of the 80C196KB. The CLKOUT pin supplies a 50% duty cycle clock signal at one-half the system clock frequency. This signal is applied to a 4-bit counter (see Appendix A) that is used to divide the frequency by 16. The resulting frequency provides one transition every eight state times which is the maximum clock frequency that should be used as a reference for the HSO.

Once initialization is complete, a wait loop is executed until the first command is received from the spacecraft. The watchdog timer is enabled and is reset each time the wait loop is executed.

B. COMMUNICATION

Packets carrying commands from the spacecraft processor will be processed by an interrupt driven routine. A flow diagram describing the algorithm to receive and validate the packet is shown in Figure 16.

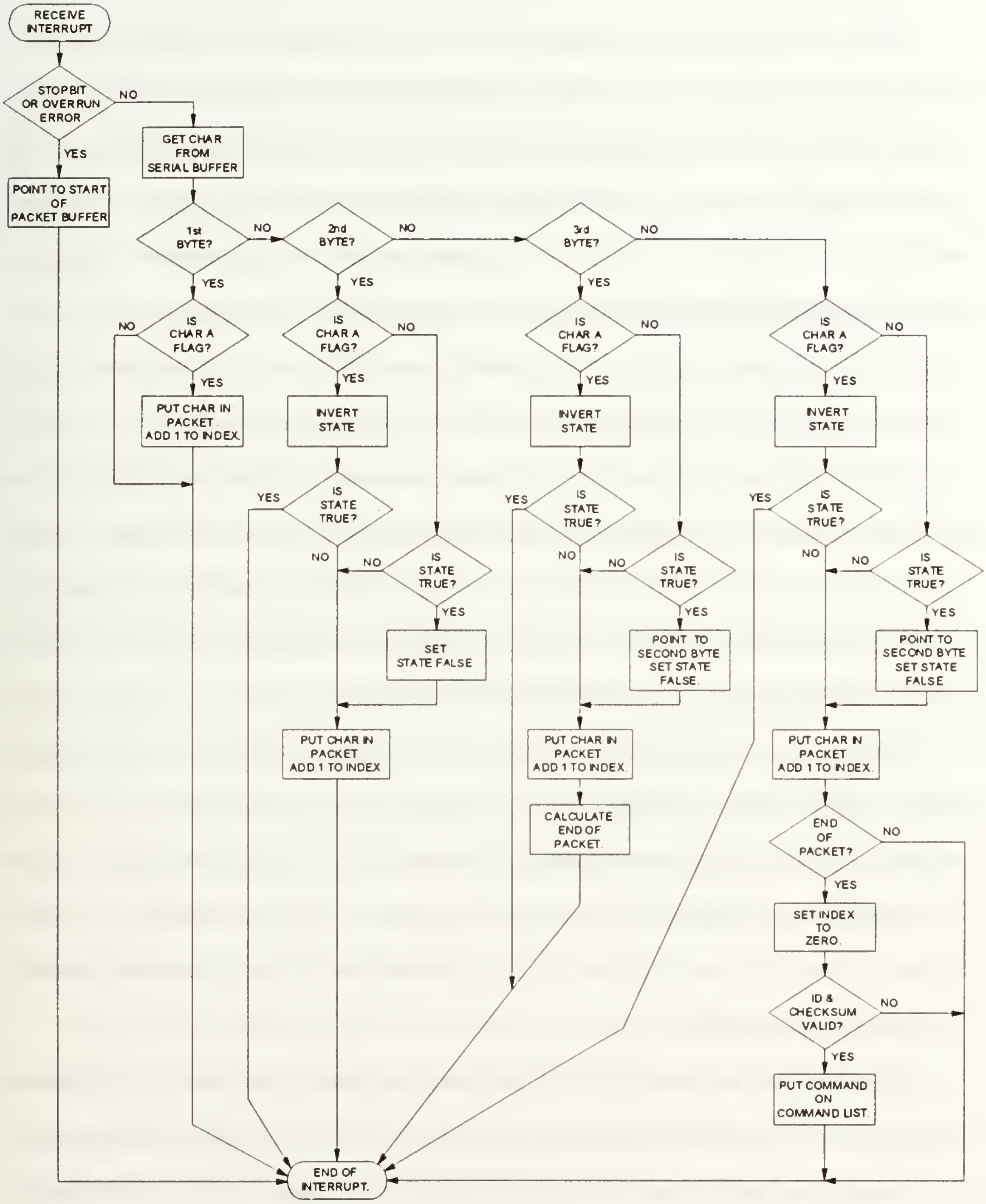


Figure 16. Receive Interrupt Flow Diagram

When the serial port receives a byte, the RI interrupt flag is set and control is transferred to the receive interrupt service routine. For all interrupt routines, the compiler generates code that disables interrupts and pushes the PSW and interrupt mask registers on the stack. The interrupts are globally disabled and the interrupt mask registers are cleared. The programmer must explicitly re-enable interrupts if they are to be serviced during the interrupt routine. Interrupts are enabled upon exiting the interrupt routine when the PSW and mask registers are restored.

The only interrupt, besides the receive interrupt, that the FERRO software uses is the software timer that updates the real-time clock. A decision was made not to enable interrupts during the receive interrupt processing due to the short time (approximately 0.2 msec average, 1 msec worst case) spent in the receive interrupt routine.

Each call of the interrupt handler will process one byte of the packet. The serial port status register is checked each time for stop bit and overrun errors. The packet is immediately discarded if any error is indicated. Once the packet is discarded, the spacecraft processor must retransmit the entire packet. If no error is detected, the byte is retrieved from the serial buffer for further processing.

The packet is assembled in a memory buffer as the individual bytes are received. An index is used to point to the next buffer location to be used. The index also indicates the position within the packet of the byte being processed.

When the index is zero, indicating the start of the packet, the routine ignores any character other than a flag character. Once the flag is received, it is stored in the buffer and the index is incremented. Flag characters occurring after the first byte must be processed to determine if they are part of a valid escape sequence. The escape sequence consists of two consecutive flag characters and is used to indicate the occurrence of a flag character within the body of a packet.

A boolean variable, *State*, is used to detect the occurrence of an escape sequence. *State* is initialized to false. When a flag character is received in the body of a packet, *State* is set to true and the routine ends, awaiting the reception of the next byte. If the next byte is a flag, *State* is again inverted; a valid escape sequence has been detected and a single flag character is entered into the buffer.

Receiving a character other than a flag when *State* is true indicates that a single flag has been encountered. When this occurs, it is assumed that the flag marks the start of a new packet. Any previously received data is discarded with the exception of the last two bytes: the flag and the byte that was just received. Since the byte just received is the second byte of the new packet, the index is set to point to the second location of the packet buffer and the byte is stored.

When the second and third bytes of the packet have been received, the length of the packet can be calculated. As mentioned previously, the size field is stored in big endian format and thus must be converted to little endian format to be used by the 80C196KB processor. The overhead length, consisting of the sum of the header and checksum lengths, is added to the converted data field length to yield the total length of the packet.

This value is used to detect when the last byte of the packet has been received. Once the entire packet has been received, the destination field is checked to ensure that it matches FERRO's address and the checksum is validated. The packet is discarded if either check fails. If the packet is valid, the command byte (i.e., the first byte of the data field) is placed in the command list to be processed. The command list is a circular queue with space for up to three pending commands.

Acknowledgment and data packets are transmitted to the spacecraft processor using a simple busy-wait loop that waits for the transmit-buffer-empty bit to be set in the serial port status register. If a flag character occurs within the body of the packet, a second flag is inserted in the data stream to indicate an escape sequence. The double-buffering feature of the serial port transmitter is used to transmit the second flag. The watchdog timer is reset after each byte is transmitted to ensure the timer does not overflow when long packets are transmitted.

C. ERROR DETECTION

The watchdog timer can detect catastrophic errors that cause a deviation from the normal program flow. During normal program operation, the software will reset the watchdog timer at intervals that will prevent the timer from overflowing. Any error that affects the program flow in a manner that prevents the reset from occurring within the normal timeframe will result in a reset of the 80C196KB microcontroller.

After the processor is reset, the initialization phase will be conducted and the experiment will have to be restarted by a command from the ground crew. If the error is not permanent in nature (i.e., due to an single-event-upset), the experiment can carry on with only a minor loss of data. A permanent fault may result in the same watchdog failure thereby precluding the continuation of the experiment.

Errors that occur during communication are detected with the two-byte checksum as specified in the OX.25 protocol. The checksum routines can also be used to detect errors that may occur during data collection. This feature was not ready in time to be integrated into the software for the experiment, but the method will be described here.

A checksum of the data buffer is calculated and stored each time data is collected. Just prior to the next data collection event the checksum validation routine is called to ensure that the data in the buffer remains unchanged. If an error is detected, a flag is set and a time tag stored so that the data in question can be identified. Another option would have been to discard the questionable data by clearing the buffer, but keeping the data and allowing the decision to be made on the ground provides for more flexibility.

D. EXPERIMENT CONTROL

Once FERRO has started, the experiment will be conducted according to a preset schedule. Scheduled events will be initiated by the experiment control routines at specified times. FERRO's real-time clock will be the reference for all scheduled events.

Overall control of the experiment will be accomplished by commands issued either by personnel on the ground or automatically by the spacecraft processor. These commands will control the initiation and shutdown of the experiment, the downloading of experiment and status data from FERRO to the spacecraft processor, and will allow for limited testing of the FERRO hardware and software without signals being applied to the ferroelectric capacitors. Table 3 shows the FERRO commands.

TABLE 3. FERRO COMMANDS [Ref. 1]

Command	Byte 1	Byte 2	Byte 3	Byte 4	Source
Data Dump	010H	*	*	*	Spacecraft
End Test Mode	020H	*	*	*	Ground
Initialize	030H	*	*	*	Ground
Restart	040H	Cycle	Cycle Day	*	Ground
Shutdown	050H	*	*	*	Ground
SOH	070H	*	*	*	Spacecraft
Test Mode	090H	*	*	*	Ground
UTime	0A0H	*	*	*	Ground/ Spacecraft
1st Parameter	0F0H	MSB Time Tag	2nd Byte Time Tag	*	Ground/ Spacecraft
2nd Parameter	0F0H	3rd Byte Time Tag	LSB Time Tag	*	Ground/ Spacecraft

* Indicates that FERRO will ignore these bytes.

1. FERRO Commands

The Initialize command is used to start the experiment and should be issued only once during the conduct of the experiment. An acknowledge packet is sent to the spacecraft processor and the command is removed from the command list. The *Cycle* and *Cycleday* variables are initialized to one and zero respectively. *Cycle* counts the number of 21-day experiment cycles conducted; *Cycleday* indicates the day within the cycle. Setting

Cycle to zero allows the first day of the cycle to begin at midnight of the day the Initialize command is sent.

The Initialize command must be followed immediately by a Utime command to set the real-time clock. Utime is really a sequence of three packets: the first specifies the Utime command, while two parameter packets are used to carry the time from the spacecraft. The time-tag value is defined as the number of seconds since January 1, 1980. The 32-bit value is sent in big endian format and thus must be converted. Only the number of seconds elapsed during the current day is needed by the experiment, so the excess is stripped using the modulus operation. The resulting value is used to set the real-time clock.

The real-time clock is implemented as a double-word (32-bit) variable that is incremented each second. The clock value indicates the number of seconds that have elapsed for the current day. Software Timer 0 is used to update the clock. Since the interrupt for this timer occurs at 0.125 second intervals, a bit is shifted through a register to count eight interrupts before incrementing the clock. The clock is reset to zero each day at midnight. Cycle and Cycleday are also updated at that time.

The Utime command is the only method to set the real-time clock. The command will be sent immediately following the Initiate and Restart commands to set the clock initially. In addition, the spacecraft processor will send the Utime command once every twenty-four hours to ensure the experiment clock remains synchronized with the spacecraft clock. When the clock is set, a critical region is established by globally disabling interrupts before altering the value. This is necessary because it takes more than one instruction to store the double-word variable.

The Restart command is similar to the Initiate command except that it is used in case the experiment was shut down or the processor was reset. The second and third bytes of the data field in the Restart packet contain values to set Cycle and Cycleday. The Restart command is issued by the ground crew who must determine where in the experiment cycle to restart the experiment. Again, the Utime command and two parameter packets must immediately follow to set the real-time clock.

After either the Initialize or Restart command has been executed, control is passed to an experiment executive control routine that manages the conduct of the experiment. The routine consists of an infinite loop that checks if a command packet has been received or if a scheduled experiment event is due to occur. Additionally, the routine initiates collection of temperature data every 32 seconds.

The Data Dump command is issued by the spacecraft processor once each day, two minutes before midnight. FERRO will respond by transmitting a packet containing the ferroelectric capacitor data collected during the day. As the polarization of the capacitors is measured according to the experiment schedule, the data is stored in a designated buffer where the data dump packet is assembled.

Although the number of measurements each day varies, the length of the packet is fixed. Separate sub-fields for fatigue and ageing data are allocated within the data field. The size of each of the sub-fields is determined by the number of measurements accomplished during the first day of an ageing cycle for either the fatigue or ageing DUTs; 23 reads of the 16 capacitors requires 368 bytes. The cycle-number and cycle-day are included in the data dump packet to identify the data.

The State of Health (SOH) command causes FERRO to respond with a packet containing experiment status and temperature data. The spacecraft processor will issue the SOH command once each minute. Data from the four temperature sensors is collected every 32 seconds and is stored the SOH packet buffer.

A flag in the SOH packet is used to indicate when the experiment is in test mode. When the SOH command is received, a time tag is obtained by reading the value of the real-time clock. The value is converted to big endian format and transmitted with the packet. More significant status reporting was planned, but was not implemented in time to be tested.

Commands to enter and exit test mode allow for limited testing of the FERRO hardware and software. While in test mode, all commands are received and acknowledged as they would be during normal operation, but signals are not applied to the DUTs. This capability allows both FERRO and the interface to the spacecraft to be tested without affecting the capacitors.

The Shutdown command is used if power must be removed from FERRO after the experiment has started. When the command is received, FERRO responds by immediately transmitting an SOH packet followed by a Data Dump packet. After sending the packets, the real-time clock is disabled to prevent further experiment events from occurring and fatiguing of the ferroelectric capacitors is stopped. The experiment can be resumed after power is restored by issuing a Restart command.

2. FERRO Experiment Events

A preset schedule of events is used to carry out the 21-day experiment cycle that is described by Figure 6 and Table 1. Scheduled events include the initiation of a fatigue cycle, writing to and reading from DUTs for ageing, and taking measurements from DUTs after periods of fatiguing. These events are scheduled to occur during the appropriate day of the experiment cycle at a time relative to the real-time clock.

When the real-time clock is updated, a list of times for events to occur during the present cycle-day is scanned and compared to the clock value. If a match is found, the event code is placed in a pending list. The experiment executive control routine will then cause the event to be executed.

A fatigue cycle must be initiated at the beginning of each 21-day experiment cycle and must be reinitiated following ageing of the fatigue DUTs. Fatiguing is started by enabling and configuring the PWM to produce the 5 kHz pulse train and setting the analog switches to route the signal to the appropriate DUTs. Fatiguing is stopped by disabling the PWM and disconnecting the DUTs.

Fatiguing of the ferroelectric capacitors in the two fatigue DUTs will be stopped briefly each day, just before the Data Dump command is expected, to collect data. Two data points will be collected for each of the sixteen capacitors. First, a positive write pulse will be applied to the capacitors followed by a read operation using a negative read pulse. The sequence will then be repeated with reversed polarities.

The HSO unit is used to generate the read and write pulses and to schedule the A/D conversion process to occur during the read pulse. Both the read and write pulses are one millisecond in duration. The pulse length was chosen to ensure that the circuitry in the path to the A/D converter has ample time to settle before the line is sampled. The write pulse could be shortened considerably, but keeping the pulses the same length had no impact on the experiment timing.

For reading the fatigue capacitors, the A/D conversion is scheduled to start 120 state times before the end of the read pulse. This ensures that the conversion process will be complete before the pulse ends. The A/D conversion is scheduled differently for the ageing read process and will be discussed below. After all of the capacitors have been read, the fatigue process is resumed.

At the start of an ageing cycle, an initial write to the DUTs being aged must be performed. After the first ageing period, the capacitors will be read and the read operation will be immediately followed by a write operation to store the value for the next ageing period. This sequence will be repeated until the ageing cycle is complete.

The read operation for ageing must be different from the one used for fatiguing because the capacitors in each DUT are connected internally on the input side. This, in conjunction with the design of the switching circuit, dictates that all eight capacitors in the DUT are subjected to any read and write pulses applied to the DUT. This does not impact the fatigue data collection since a write operation is performed immediately before each read operation. Each capacitor is written to and read from twice; thus 32 pulses are applied to all eight capacitors in the DUT although only four are pertinent to each one. These few extra read and write pulses applied to the capacitors are insignificant compared to the number of fatigue pulses applied.

The problem with using a separate read pulse for each capacitor for ageing measurements is that the read pulse for one capacitor will effectively act as a write pulse to the other capacitors in the DUT. These capacitors will be repolarized in the direction of the read pulse and thus the ageing data would be invalid for all but the first capacitor read. The polarization measured from these capacitors would be the result of the previous capacitor's read pulse rather than the write operation performed at the beginning of the ageing period.

The solution to this problem is to apply a single read pulse to the DUT and conduct all eight A/D conversions in sequence during the pulse. The CAM is not large enough to schedule all eight A/D conversions, so only the first is scheduled and a busy-wait loop is used to conduct the remaining seven conversions in succession.

E. TEST/VERIFICATION

An IBM PC based software program designed to simulate the spacecraft processor was used for testing during FERRO hardware and software development and to verify the functional operation of FERRO when the experiment was delivered for integration to the spacecraft bus. The program uses the PC's RS-232 serial port for communication with FERRO via RS-232 to RS-422 level shifters.

The program was developed using the C compiler included with Borland C++. The compiler offers access to the operating system and PC BIOS calls necessary to directly control the serial interface. The integrated development environment offers a program editor, project manager, and source-level debugger. A windowing and menu library was used to provide a user-friendly interface for the program. The source code for the test and verification software is included as Appendix C.

Pull down menus provide selections to set communication parameters including baud rate, number of data bits, parity, and number of stop bits. These parameters can be saved in a configuration file so that future sessions will default to the chosen setup. Other selections initialize or reset the serial communication port, allow recording of all communications in a log file, and send command packets to the FERRO experiment.

All FERRO commands can be sent by the software and prompts are provided for the user to supply data for the Parameter packets when setting the real-time clock. The outgoing packet is displayed in one window and the response from FERRO is displayed in another. The packet is parsed and labeled for display so that the packet header, data field, and checksum can be distinguished. The status of the checksum validation is displayed following the checksum field.

A record of all communication between FERRO and the test/verification software can be saved in a log file. A menu selection toggles the logging feature on or off. The user can select a file name for the log file or use the default file name.

IV. CONCLUSIONS AND RECOMMENDATIONS

Several goals set at the beginning of software development were only partially met. The most significant shortfall was in the area of error detection. Additionally, a complete test of the FERRO software on the experiment hardware was not completed prior to delivery of the experiment.

When software development for FERRO began, the specifications for the communication interface between the experiment and the spacecraft processor were immature in several areas including the command structure, universal time format, and use of the control field. Software development had to proceed before these specifications could be firmed up to meet the schedule for delivery of the experiment.

Additionally, several specifications including the length of the size, source, and destination fields were changed well into the software development cycle. These factors resulted in extensive changes to routines that were already functional and tested. A significant amount of time was spent making these changes and debugging the inevitable errors introduced by the changes. As a result, several planned features were not implemented in the FERRO software. It ultimately became more important to get something that worked out the door rather than add and test additional features.

A method for performing checksum validation of program memory and the data collection buffers periodically would ensure that SEUs and hard faults caused by radiation exposure do not go undetected. This error checking in conjunction with storing multiple copies of the program code in

ROM would allow the experiment to suffer multiple faults and still be able to complete the mission.

The checksum validation routine used for communication was designed so that it could also be used to check either program memory or the data collection buffers. The proposed method for checking the data buffers was discussed earlier. To implement error checking for the program code, a checksum for a specified area of program memory would have to be calculated off line and stored in ROM at the end of the program code. Periodically, the checksum validation routine would be called to validate the program memory.

If an error was detected, a flag would be set and a system reset conducted. The program start-up code would recognize the flag (registers are not cleared by a reset) and would call a different copy of the experiment software. There would be areas of the software that could not be duplicated (i.e., the start-up code), but this type of error checking and recovery mechanism could significantly increase reliability of the experiment software.

Another area that could be improved is the command interface. The command structure evolved through the many specification changes and the result is not as clean and efficient as originally envisioned. One possible improvement would be to implement the command decision tree with a state matrix. Rather than ignoring illegal commands, a flag could be set in the acknowledge packet so that the that the spacecraft processor would not simply retransmit the command. More status and mode information should be included in the SOH packet. Additional information could be used by ground personnel to take appropriate corrective action should problems occur.

Finally, the format of the data packet could be improved to make it easier to extract and reconstruct the experiment data to conduct analysis. A variable length data packet with tag or label fields for each group of capacitor data would be more efficient and would make it easier to positively identify the data.

LIST OF REFERENCES

1. Orbital Sciences Corporation, *FERRO/APEX Interface Control Document*, May 1992.

2. Bondurant, David and Gnadinger, Fred, "Ferroelectrics for Nonvolatile RAMs," *IEEE Spectrum*, v. 26, pp 30-33, July 1989.

3. Paz de Araujo, C. E., and Scott, J. F., "Ferroelectric Memories," *Science*, v. 246, pp. 1400-1405, December 1989.

4. William Rives Hestir, *Automated Testing of On-chip Ferroelectric Capacitors*, M.S. Thesis, Naval Postgraduate School, CA, December 1990.

5. Jonker, G. H., "Nature of Ageing in Ferroelectric Ceramics," *Journal of the American Ceramic Society*, v. 55, pp 57-58.

6. Sawyer, C. B., and Tower, C. H., "Rochelle Salt as a Dielectric," *Physical Review*, v. 35, pp. 269-273, 1930.

7. Bondurant, David, "Ferroelectric RAM Memory Family for Critical Data Storage," *Proceedings of the First Symposium on Integrated Ferroelectrics*, Colorado Springs, CO, pp. 212-215, March 1989.

8. Matthew Raycroft Kercher, *Design of an Autonomous Test Device for Ferroelectric Components*, M.S. Thesis, Naval Postgraduate School, CA, June 1991.

9. Intel Corporation, *16-Bit Embedded Controller Handbook*, 1990.

10. Orbital Sciences Corporation, *OSC OX.25 Protocol Document E60001*, May 1989.

11. Intel Corporation, *iC-96 Compiler User's Guide for DOS Systems*, 1990.

12. Annapolis Micro Systems, *Environment 96*. 1990.

13. Intel Corporation, *Embedded Applications*, 1990.

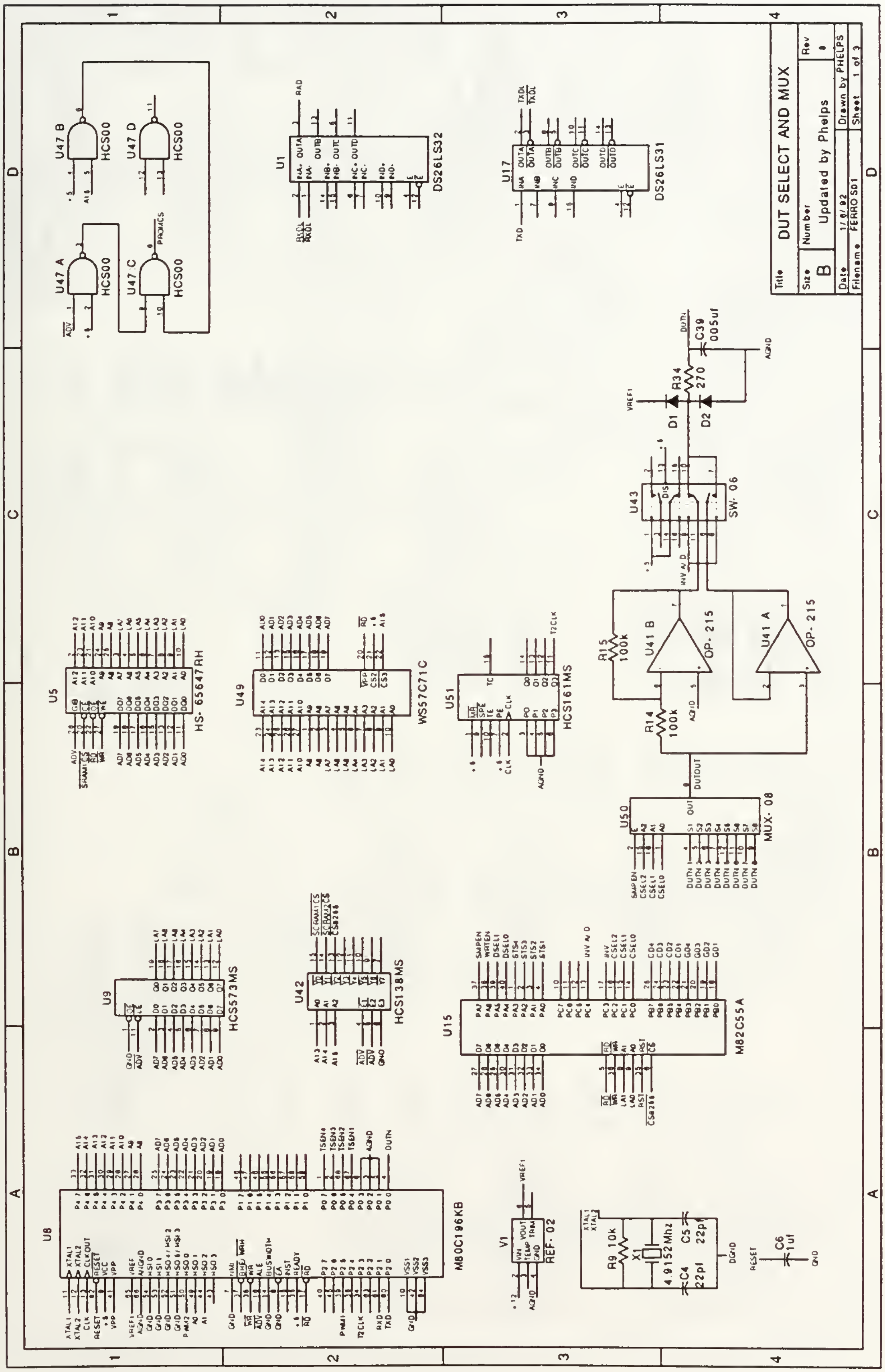
14. Katz, Ron and Boyet, Howard, *The 16-Bit 8096: Programming, Interfacing, Applications*, Microprocessor Training Inc, 1991.

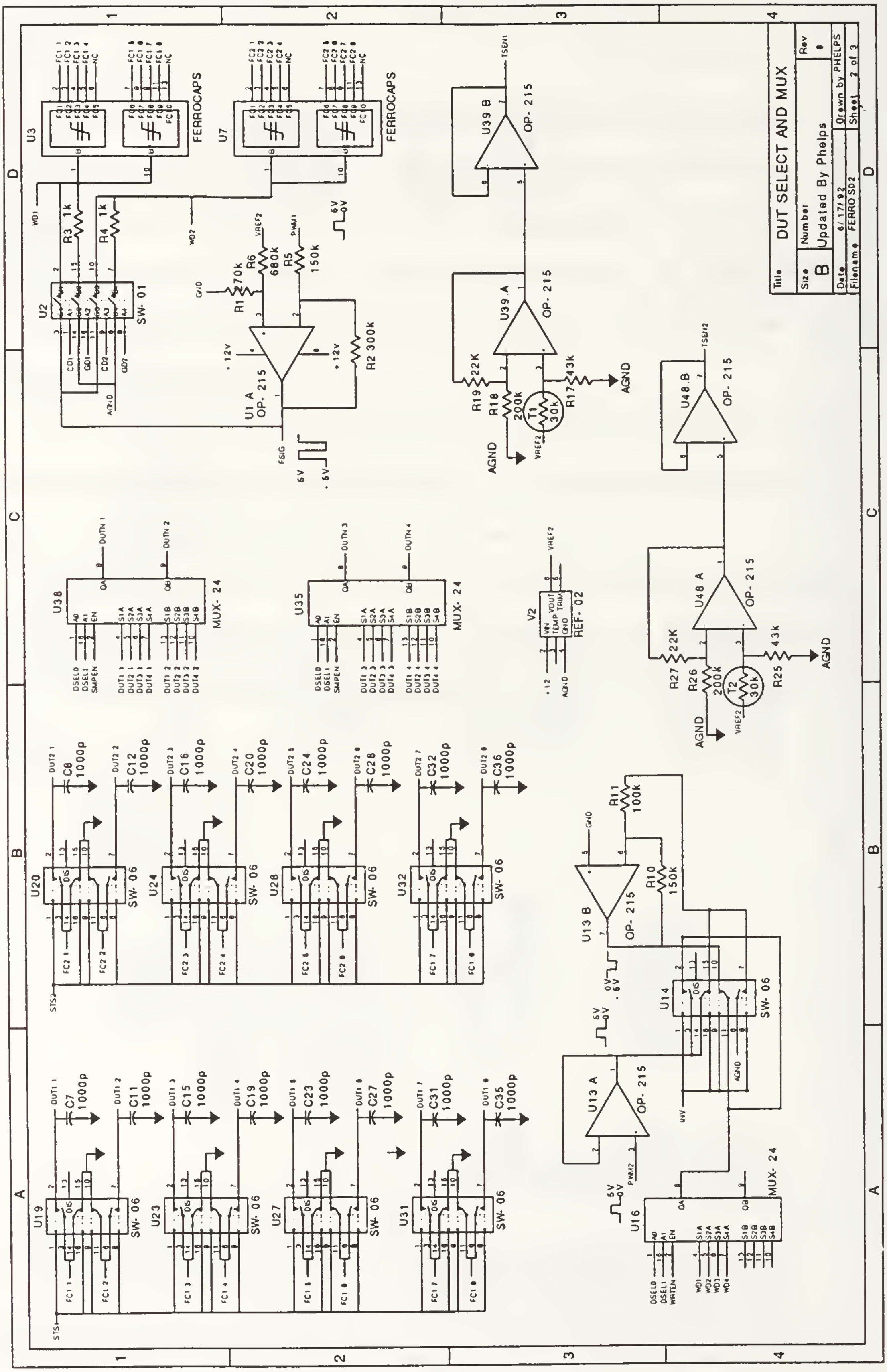
APPENDIX A - FERRO EXPERIMENT HARDWARE SCHEMATIC

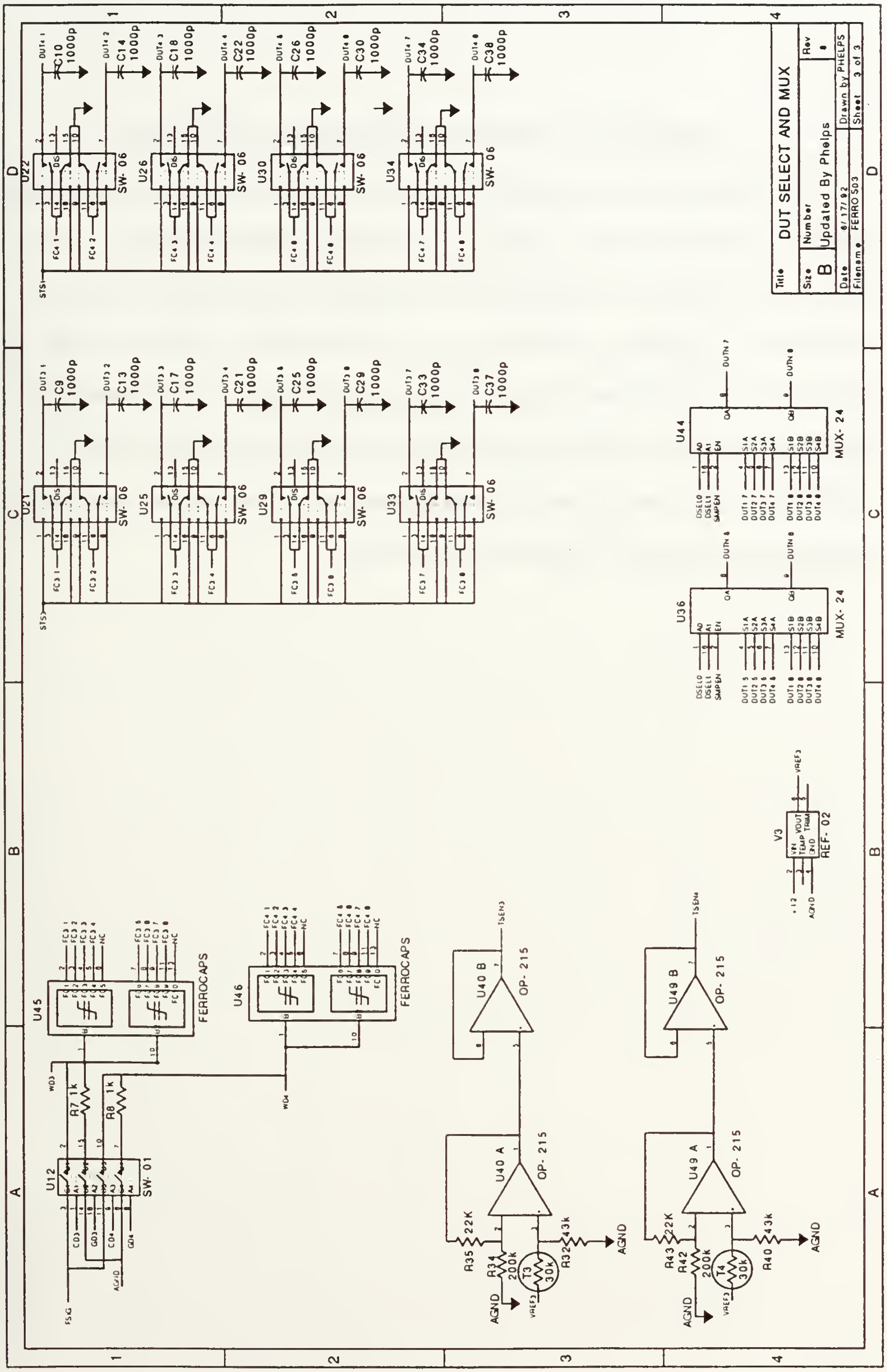
This appendix contains the schematic for the FERRO hardware. The 80C196KB Microcontroller and the 82C55 Programmable Periperheral Interface are shown on sheet 1. Also shown on sheet 1 is the chip select logic, the clock circuitry and the A/D converter input multiplexor for reading the ferroelectric capacitors.

Sheet 2 shows the driver amplifiers and switching for applying read, write, and fatigue pulses to the capacitors. The sense capacitors and Sawyer-Tower switches for DUTs 1 and 2 are also shown.

Sheet 3 shows the sense capacitors and Sawyer-Tower switches for DUTs 3 and 4. The temperature sensing circuitry is split between sheets 2 and 3 as are the DUT select multiplexers for reading the ferroelectric capacitors.


DUT SELECT AND MUX
Title
Size
B
Number
Updated by Phelps
Rev
0
Date
1/6/92
Drawn by PHELPS
Filename
FERROSD1
Sheet
1 of 3
U47 A
U47 B
U47 C
U47 D
HCS00
U1
DS26LS32
U17
DS26LS31
U5
HS-65647RH
U49
WS57C71C
U51
HCS161MS
U41 A
U41 B
OP-215
R14
100k
R15
100k
U43
SW-06
D1
D2
R34
270
C39
.005uf
U50
MUX-08
U9
HCS573MS
U42
HCS138MS
U15
M82C55A
U8
M80C196KB
V1
REF-02
R9 10k
X1
4.9152Mhz
C4
22pf
C5
22pf
C6
1uf


DUT SELECT AND MUX
Title
Size B
Number
Updated By Phelps
Rev 0
Date 6/17/92
Drawn by PHELPS
Filename FERRO S02
Sheet 2 of 3
U3 FERROCAPS
U7 FERROCAPS
U2 SW-01
R3 1k
R4 1k
R1 270k
R6 680k
R5 150k
R2 300k
U1 A OP-215
FSIG
6V
-6V
U39 A OP-215
U39 B OP-215
TSEN1
R19 22K
R18 200k
T1 30k
R17 43k
AGND
VREF2
U48 A OP-215
U48 B OP-215
TSEN2
R27 22K
R26 200k
T2 30k
R25 43k
U38 MUX-24
U35 MUX-24
DUTN 1
DUTN 2
DUTN 3
DUTN 4
V2 REF-02
U20 SW-06
U24 SW-06
U28 SW-06
U32 SW-06
U19 SW-06
U23 SW-06
U27 SW-06
U31 SW-06
C8 1000p
C12 1000p
C16 1000p
C20 1000p
C24 1000p
C28 1000p
C32 1000p
C36 1000p
C7 1000p
C11 1000p
C15 1000p
C19 1000p
C23 1000p
C27 1000p
C31 1000p
C35 1000p
STS
U13 A OP-215
U13 B OP-215
R11 100k
R10 150k
U14 SW-06
U16 MUX-24
PWM2
DSEL0
DSEL1
WRTEN
WD1
WD2
WD3
WD4
INV


Title DUT SELECT AND MUX
Size B
Number Updated By Phelps
Rev 8
Date 6/17/92
Drawn by PHELPS
Filename FERRO S03
Sheet 3 of 3
U22 SW-06
U26 SW-06
U30 SW-06
U34 SW-06
U21 SW-06
U25 SW-06
U29 SW-06
U33 SW-06
C10 1000p
C14 1000p
C18 1000p
C22 1000p
C26 1000p
C30 1000p
C34 1000p
C38 1000p
C9 1000p
C13 1000p
C17 1000p
C21 1000p
C25 1000p
C29 1000p
C33 1000p
C37 1000p
U36 MUX-24
U44 MUX-24
DSEL0
DSEL1
SMPEN
V3 REF-02
VREF3
U12 SW-01
U45 FERROCAPS
U46 FERROCAPS
R7 1k
R8 1k
U40 A OP-215
U40 B OP-215
U49 A OP-215
U49 B OP-215
R35 22K
R34 200k
T3 30k
R32 43k
R43 22K
R42 200k
T4 30k
R40 43k
AGND
TSEN3
TSEN4
FSIG

APPENDIX B - FERRO SOFTWARE SOURCE CODE

The source code for the FERRO software is divided in to several files based on functional area. FERRO.H contains all global definitions and macros and all function prototypes. All global variables are declared in GLOBALS.H. INIT96.C contains all of the initialization routines, while the bulk of the command and event processing routines are in COMMAND.C. The communication routines are located in PACKET.C and CHECK.C. EXPER.C contains all of the routines to read and write to the ferroelectric capacitors and read the temperature sensors.

```
                           /* C96INIT.H */

#pragma model(kb)          /* allow KB SFR's & instr to be used */
#pragma pagewidth 80
#pragma pagelength 58
#pragma tabwidth  3
#include <80c196.h>        /* definitions of 196 SFR's */
```

```
                            /* FERRO.H */

#pragma regconserve                          /* Don't use registers for variables    */

/***************************** PACKET DEFINITIONS ******************************/
#define FLAG            0x7E                 /* Packet begin flag                    */
#define NPSID_HI        0x4D                 /* high byte of FERRO id                */
#define NPSID_LO        0x01                 /* low byte of FERRO id                 */
#define HEAD_LEN        9                    /* Length of packet header              */
#define OVERHEAD        HEAD_LEN+2           /* Header plus checksum                 */
#define MAX_PACKET      OVERHEAD+1000        /* Length of maximum size packet        */
#define PLEN            1                    /* Index of length field of packet      */
#define SRC_FLD         5                    /* Index for source field               */
#define DEST_FLD        3                    /* Index for destination field          */
#define CTRL_FLD        7                    /* Index for control field              */
#define FALSE           0
#define TRUE            1

#define SOHSIZE         39                   /* # of data bytes in SOH packet        */
#define ACKSIZE         1                    /* # of data bytes in ACK packet        */
#define DMPSIZE_HI      0x02                 /* HI byte of data dump size field      */
#define DMPSIZE_LO      0xE4                 /* LO byte of data dump size field      */


/***************************** DUT ASSIGNMENTS *********************************/
#define FATDUT1    0                         /* inside                               */
#define FATDUT2    2                         /* outside                              */
#define AGEDUT1    1                         /* inside                               */
#define AGEDUT2    3                         /* outside                              */


/***************************** EXPERIMENT EVENT DEFINITIONS ********************/
#define MAXSECONDS      0xFFFFFFFF           /* Constant used to end event schedules */
#define RD_FAT_TIME     86250                /* 2 & 1/2 minutes before 00:00 hours   */
#define WRT_AGE1        1                    /* Event codes                          */
#define WRT_AGE2        2
#define RD_AGE1         3
#define RD_AGE2         4
#define RD_FAT          5
#define STRT_FAT        6
#define NULLEVENT       0
#define INV             1

/***************************** EXPERIMENT CLOCK DEFINITIONS ********************/
#define TICKPERIOD      0x9600                /* Timer 2 count for .125 seconds       */
#define ONEDAY          86400L                /* # of seconds in one day              */

/***************************** EXPERIMENT HSO PULSE DEFINITIONS ****************/
#define PULSELEN   313                        /* Timer1 count for 1 msec              */
#define STARTHI    5                          /* Count for start of positive pulse    */
#define ENDHI      STARTHI+PULSELEN           /* Timer2 count for HSO.0 to go low     */
#define FIRST_AD   STARTHI+75                 /* 20 microsecond delay                 */
#define AD_TIME    ENDHI-15                   /* A/D conv needs 91 states             */

/***************************** BIT MASKS ***************************************/
#define AD_COMP         0x01     /* Bit # for  A/D Conv complete interrupt bit */
#define CLR_AD_COMP     0xFD     /* Mask to clear A/D Conv complete bit        */
#define TXE             0x08     /* Bit value for xmitter empty in sp_stat     */
#define ACK_RESPNS      0x08     /* ACK Response bit in packet control field   */
#define TEMPMASK        0x1F     /* Mask seconds to get temp every 32 seconds  */
#define HALFSEC         0x10     /* Mask for tick at approx .5 second point    */

/***************************** MACROS ******************************************/
#define WAIT_FOR_CAM asm nop; asm nop; asm nop; asm nop
#define POPCMD       nextcmd = (nextcmd+1) & 0x03
```

```
#define ROUND         ((ad_result_lo & 0x80) ? 1 : 0)
/**************************** EXPERIMENT COMMAND CODES *************************/
#define INITIALIZE   0x30
#define DATADUMP     0x10
#define SOH          0x70
#define TESTMODE     0x90
#define ENDTESTMODE  0x20
#define SHUTDOWN     0x50
#define RESTART      0x40
#define UTIME        0xA0
#define PARAMETER    0xF0

/******************************** TIME UNION ***********************************/
/* used to convert GPS time to internal 32-bit variable storage format      */
/* (i.e. little endian).                                                     */

typedef union {
                unsigned long secs;
                unsigned char buf[4];
               } TIME;

/******************************** PACKET STRUCTURES ****************************/
typedef struct  {
                   unsigned char flag;       /* 0x7E */
                   unsigned char size_hi;    /* size high byte */
                   unsigned char size_lo;    /* size low byte */
                   unsigned char dest_hi;    /* source field */
                   unsigned char dest_lo;
                   unsigned char src_hi;     /* destination field */
                   unsigned char src_lo;
                   unsigned char ctrl_hi;    /* control field */
                   unsigned char ctrl_lo;
                  } HEADER;

typedef struct  {
                        HEADER head;
                        unsigned char temp[4];
                        unsigned char status;
                        unsigned char cycle;
                        unsigned char cycleday;
                        unsigned char time[4];          /* gps time */
                        unsigned char cap_tol[28];      /* spare */
                        unsigned char csum[2];
                      } SOHPKT;

typedef struct  {                                  /* ack packet structure */
                        HEADER head;
                        unsigned char ack_cmd;
                        unsigned char csum[2];
                      } ACKPKT;

typedef struct  {
                        HEADER head;
                        unsigned char cycle;
                        unsigned char cycleday;
                        unsigned char fdata[368];
                        unsigned char adata[368];
                        unsigned char csum[2];
                     }DATAPKT;

/******************************** FUNCTION PROTOTYPES **************************/
/* init96.c */
void main(void);
void init96(void);
void wdog(void);
void timertick(void);
void init_timer(void);
void convert_time(void);
void clr_buf(void);
void clr_soh(void);

/* command.c */
void run_exper(void);
void send_soh(void);
void send_ack(unsigned char cmd);
void send_dump(unsigned char *dhead);
void do_event(void);
void wait_for_command(unsigned char);
void set_utime(void);
void read_time(void);   /* new not tested */

/* exper.c */
void startfatigue(void);
void stopfatigue(void);
void write_pulse(void);
void read_pulse(void);
void age_read_pulse(void);
void get_temps(void);
void read_ageing(unsigned char dut_sel, unsigned char *bufptr);
void getfatigue(unsigned char dut, unsigned char fcap, unsigned char * bufptr);
void write_dut(unsigned char dut_sel);

/* packet.c */
void rcv_packet(void);
void send_packet(unsigned char *buf);

/* check.c */
void compute_checksum(unsigned char *buf);
int valid_csum(unsigned char *packet, unsigned int len);
void test(void);
```

```
                              /* GLOBALS */
/****************************************************************************/
/* Each variable length array contains the a list of scheduled times for   */
/* the events to occur for that day.  The end of the list is marked by     */
/* the constant MAXSECONDS                                                  */
/****************************************************************************/
const long day0[] = {MAXSECONDS};
const long day1[] = {0,1,3,6,10,15,
                     25,45,75,115,165,
                     265,465,765,1165,1665,
                     2665,4665,7665,11665,16665,
                     26665,46665,76665,RD_FAT_TIME,MAXSECONDS};
const long day2[] = {30265,80265,RD_FAT_TIME,MAXSECONDS};
const long day3[] = {RD_FAT_TIME,MAXSECONDS};
const long day4[] = {7465,RD_FAT_TIME,MAXSECONDS};
const long day5[] = {RD_FAT_TIME,MAXSECONDS};
const long day6[] = {34665,RD_FAT_TIME,MAXSECONDS};
const long day7[] = {0,1,3,6,10,15,
                     25,45,75,115,165,
                     265,465,765,1165,1665,
                     2665,4665,7665,11665,16665,
                     26665,46665,76665,MAXSECONDS};
const long day8[] = {0,RD_FAT_TIME,MAXSECONDS};
const long day9[] = {75465,RD_FAT_TIME,MAXSECONDS};
const long day10[] = {RD_FAT_TIME,MAXSECONDS};
const long day11[] = {RD_FAT_TIME,MAXSECONDS};
const long day12[] = {RD_FAT_TIME,MAXSECONDS};
const long day13[] = {RD_FAT_TIME,MAXSECONDS};
const long day14[] = {0,1,3,6,10,15,
                     25,45,75,115,165,
                     265,465,765,1165,1665,
                     2665,4665,7665,11665,16665,
                     26665,43465,46665,76665,MAXSECONDS};
const long day15[] = {0,RD_FAT_TIME,MAXSECONDS};
const long day16[] = {RD_FAT_TIME,MAXSECONDS};
const long day17[] = {RD_FAT_TIME,MAXSECONDS};
const long day18[] = {RD_FAT_TIME,MAXSECONDS};
const long day19[] = {RD_FAT_TIME,MAXSECONDS};
const long day20[] = {25065,RD_FAT_TIME,MAXSECONDS};
const long day21[] = {0,1,3,6,10,15,
                     25,45,75,115,165,
                     265,465,765,1165,1665,
                     2665,4665,7665,11665,16665,
                     26665,46665,76665,MAXSECONDS};

/****************************************************************************/
/* Forms an array of the schedule lists for the 21 days of the experiment */
/* cycle.                                                                   */
/****************************************************************************/
const long * const sched[] = {day0, day1, day2, day3, day4, day5,
                              day6, day7, day8, day9, day10, day11,
                              day12, day13, day14, day15, day16, day17,
                              day18, day19, day20, day21};

/****************************************************************************/
/* This lists of events match up with the lists of scheduled times.  When */
/* a scheduled time has been reached, the corresponding entry in the list */
/* for the appropriate day is referenced to determine the proper event to */
/* execute.                                                                 */
/****************************************************************************/

const unsigned char event0[] = {NULLEVENT};
const unsigned char event1[] = {WRT_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,

                                RD_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,
                                RD_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,
```

```
                                RD_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,
                                RD_AGE1,RD_AGE1,RD_AGE1,RD_AGE1,RD_FAT};
const unsigned char event2[] = {RD_AGE1,RD_AGE1,RD_FAT};
const unsigned char event3[] = {RD_FAT};
const unsigned char event4[] = {RD_AGE1,RD_FAT};
const unsigned char event5[] = {RD_FAT};
const unsigned char event6[] = {RD_AGE1,RD_FAT};
const unsigned char event7[] = {WRT_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,

                                RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,
                                RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,
                                RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,
                                RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2};
const unsigned char event8[] = {STRT_FAT,RD_FAT};
const unsigned char event9[] = {RD_AGE1,RD_FAT};
const unsigned char event10[] = {RD_FAT};
const unsigned char event11[] = {RD_FAT};
const unsigned char event12[] = {RD_FAT};
const unsigned char event13[] = {RD_FAT};
const unsigned char event14[] = {WRT_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2
,
                                 RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,
                                 RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,
                                 RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,
                                 RD_AGE2,RD_AGE1,RD_AGE2,RD_AGE2,};
const unsigned char event15[] = {STRT_FAT,RD_FAT};
const unsigned char event16[] = {RD_FAT};
const unsigned char event17[] = {RD_FAT};
const unsigned char event18[] = {RD_FAT};
const unsigned char event19[] = {RD_FAT};
const unsigned char event20[] = {RD_AGE1,RD_FAT};
const unsigned char event21[] = {WRT_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2
,
                                 RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,
                                 RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,
                                 RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,RD_AGE2,
                                 RD_AGE2,RD_AGE2,RD_AGE2};


/****************************************************************************/
/* Forms an array of the event lists for the 21 days of the experiment    */
/* cycle.                                                                   */
/****************************************************************************/

const unsigned char * const event[] = {event0, event1, event2, event3,
                                       event4, event5, event6, event7,
                                       event8, event9, event10, event11,
                                       event12, event13, event14, event15,
                                       event16, event17, event18, event19,
                                       event20, event21};

/* Declaration of structure types defined in FERRO.H                      */
SOHPKT soh_buf;

ACKPKT ack_buf;

DATAPKT dmp_buf;

TIME GPS;                            /* temp storage for converting GPS time */

unsigned int FATIGUE_CAPS;        /* flag 4 dis/enabling fatiguing of fcaps  */

unsigned char rbuf[MAX_PACKET];      /* buffer to hold received packet       */
register unsigned char eobuf;        /* index of end of packet on receive    */
register unsigned char rcv_ptr;      /* points to next byte in rbuf on rcv   */
register unsigned char state;     /* keeps track of double FLAGs within pkt */
register char tick;                /* register to count 8 ticks for 1 sec  */
unsigned long seconds;            /* experiment clock (secs since midnite) */
unsigned char cycle;               /* # of cycles since beginning of exper */
unsigned char cycleday;            /* day within a cycle (1 thru 21)       */

unsigned char nextevent;           /* event code to be executed            */
unsigned char cmdlist[4];          /* circular buffer to store cmd         */
unsigned char nextcmd;             /* ptr to next empty spot in cmd list   */
unsigned char cmdptr;              /* ptr to next cmd in list to execute   */

unsigned char agereadcycle1;       /* # of age reads executed today        */
unsigned char agereadcycle2;       /* 1 = ageing fcaps;  2 = fatigue fcaps */

/* The 8255 I/O ports and control register are declared and mapped to      */
/* memory locations here.                                                  */

unsigned char porta, portb, portc, ctl_8255;
#pragma locate(porta=0x0C000, portb=0x0C001)
#pragma locate(portc=0x0C002, ctl_8255=0x0C003)

/* DUT and Sawyer-Tower select values for reading DUTs 1,2,3, or 4 */
const unsigned char rd_dut_val[4] = {0x41, 0x52, 0x64, 0x78};

/* DUT select values for writing a value to DUTs 1,2,3, or 4 */
const unsigned char wrt_dut_val[4] = {0x40, 0x50, 0x60, 0x70};

/* Define checksum bytes to check code in ROM for errors */
const unsigned char check0 = 0;
const unsigned char check1 = 0;
#pragma locate(check0=0x03346, check1=0x03347)
```

```
                                  /* INIT96.C */
#include "ferro.h"
#include "globals.h"

#pragma CCB(0xF0)
#pragma interrupt (timertick=5)     /* set tick to handle soft tmr interrupt */

/**************************************************************************/
/* This function calls the initialization routines and waits for one of 3  */
/* commands.  The INITIALIZE command starts the experiment.  It must be    */
/* followed by a UTIME command with its 2 parament packets to set the  to  */
/* experiment clock.  The cycle # is set to 1 and cycleday to 0.  This     */
/* command is expected to be executed only once.  If the experiment must   */
/* be restarted, the RESTART command should be received.  The experiment   */
/* cycle and cycleday are received in the command packet.  This command    */
/* must also be followed by the UTIME command.  Both of these commands     */
/* transfer control to the run_exper function which will process           */
/* subsequent commands.  If the TESTMODE command is received, a flag is    */
/* changed to prevent application of pulses to the fcaps.                  */
/**************************************************************************/

void main(void)
{
   int i;
   extern unsigned int FATIGUE_CAPS;
   extern unsigned char rbuf[];

   init96();
   cmdptr = 0;
   nextcmd = 0;
   clr_buf();                  /* clear data buffer                       */
   clr_soh();                  /* clear soh buffer                        */
   while(TRUE)
      {
      while(cmdptr == nextcmd)            /* spin while no command         */
         wdog();

      switch(cmdlist[nextcmd])
         {
         case INITIALIZE:  POPCMD;
                           if (rbuf[CTRL_FLD] & ACK_RESPNS)
                              send_ack(INITIALIZE);
                           cycle = 1;
                           cycleday = 0;
                           nextevent = NULLEVENT;
                           init_timer();      /* start the experiment clock */
                           wait_for_command(UTIME);
                           set_utime();       /* set the clock */
                           run_exper();
                           break;

         case RESTART:     POPCMD;       /* cycle & day sent as part of cmd */
                           if (rbuf[CTRL_FLD] & ACK_RESPNS)
                              send_ack(RESTART);
                           cycle = rbuf[HEAD_LEN+1];
                           cycleday = rbuf[HEAD_LEN+2];
                           init_timer();    /* restart the experiment clock */
                           wait_for_command(UTIME);
                           set_utime();          /* set the clock */
                           nextevent = WRT_AGE1; /* wrt initial vals to fcaps */
                           do_event();
                           run_exper();
                           break;

         case TESTMODE:    POPCMD;       /* stays in this case after received */
```

```
                           if (rbuf[CTRL_FLD] & ACK_RESPNS)
                              send_ack(TESTMODE);
                           FATIGUE_CAPS=FALSE;
                           soh_buf.status=0;
                           break;

         default:          POPCMD;       /* ignore all other commands */
                           break;
         }
      }
}

/**************************************************************************/
/*  This function initializes the onboard peripherals of the 80C196KB and */
/*  initializes variables and pointers.                                   */
/**************************************************************************/

void init96(void)
{
   ctl_8255 = 0x80;            /* 8255 ports A, B, and C are output ports   */
   porta = 0;                  /* initialize 8255 ports                     */
   portb = 0xF0;
   portc = 0;
   baud_rate = 0x1f;           /* count for 9600 baud (low byte)            */
   baud_rate = 0x80;           /* (hi byte)-- use XTAL1 as clk source       */
   ioc1 = 0x20;                /* enable TXD vice P2.0                      */
   ioc2 = 0xD4;                /* PWM Prescaler on, A/D Prescaler off       */
   sp_con = 0x09;              /* rcv enable, no parity, mode 1             */
   int_mask = 0x20;            /* software timer interrupt enabled          */
   imask1 = 0x02;              /* receiver interrupt enabled                */
   state = FALSE;              /* toggle for FLAGs embedded in packets      */
   rcv_ptr = 0;                /* point to start of receive buffer          */
   FATIGUE_CAPS = TRUE;        /* false == testmode ie no fatigue of caps   */
   soh_buf.status = 1 ;        /* Not in test mode                          */
   asm ei;                     /* enable interrupts                         */
}

/**************************************************************************/
/* This function resets the watchdog timer when called.                   */
/**************************************************************************/

void wdog(void)
{
   watchdog = 0x1E;
   watchdog = 0xE1;
}

/**************************************************************************/
/* Timertick is the interrupt handler for the HSO Software Timer 0        */
/* interrupt. Software Timer 0 will cause an interrupt every 0.125        */
/* seconds.  The tick variable is initialize with the LSB set.  Each      */
/* time the interrupt routine is called the bit is shifted one position   */
/* to the left.  On the 8th shift, the carry-out flag is set, resulting   */
/* in reinitializing the tick variable, incrementing the seconds variable, */
/* updating the cycle day variable, and checking the event schedule.  The */
/* seconds variable counts the number of seconds elapsed during the day.  */
/* If an event is scheduled to occur on that second, the event code is    */
/* retreived and inserted as the nextevent.                               */
/**************************************************************************/

void timertick(void)
{
   unsigned char i;
   extern const long * const sched[];
   extern const unsigned char * const event[];
   extern unsigned long seconds;
   extern unsigned char cycleday, nextevent;
   extern unsigned char agereadcycle1, agereadcycle2;

   asm shlb tick, #1;
   asm bnc  ENDTICK;              /* jump if 8 ticks haven't occurred      */
   asm ldb  tick, #0x01;          /* initial value to be shifted each tick */
   seconds++;

   if(seconds == ONEDAY)
      {
      seconds = 0;
      cycleday = (cycleday == 21) ? 1 : cycleday + 1;
      agereadcycle1 = 0;
      agereadcycle2 = 0;
      }

   imask1 = 0x02;                 /* allow for rcv interrupt after this    */
   asm ei;

        /* Search the event schedule to see if its time to do event           *
/
   for(i=0; sched[cycleday][i] < seconds; i++);
   if(sched[cycleday][i] == seconds)
      {
      nextevent = event[cycleday][i];
      }
   else
      nextevent = NULLEVENT;

ENDTICK:
   return;
}

/****************************************************************************/
/*  This function initializes the tick variable with the LSB set and sets  */
/*  up the HSO Software timer 0 to expire and initiate an interrupt every  */
/*  0.125 seconds.  Timer 2 is used as the reference and is reset by the   */
/*  HSO unit each tick.                                                     */
/****************************************************************************/

void init_timer(void)
{
   tick = 0x01;                   /* initial value to be shifted each tick */
   seconds = 0;
   ioc2 |= 0x80;                  /* make sure the CAM is clear to start   */
   hso_command = 0xEE;            /* reset timer2 each tick, locked in CAM */
   hso_time = TICKPERIOD;
   WAIT_FOR_CAM;
   hso_command = 0xF8;            /* Software timer 0 locked in CAM        */
   hso_time = TICKPERIOD;
}

/****************************************************************************/
/*  This function sets each byte of the data packet to zero.               */
/****************************************************************************/

void clr_buf(void)
{
   extern DATAPKT dmp_buf;
   int i;

   for(i=0; i < sizeof(DATAPKT); i++)
      ((unsigned char *) &dmp_buf)[i] = 0;
}

/****************************************************************************/
```

```
/*  This function sets each byte of the SOH packet to zero.                */
/****************************************************************************/

void clr_soh(void)
   {
   extern SOHPKT soh_buf;
   int i;

   for(i=0;i<=3;soh_buf.temp[i++]=0);
   soh_buf.status = 0;
   soh_buf.cycle = 0;
   soh_buf.cycleday = 0;
   for(i=0;i<=3;soh_buf.time[i++] = 0);
   for(i=0;i<=27;soh_buf.cap_tol[i++] = 0);
   }
```

```
                             /* COMMAND.C */

#include "ferro.h"

/****************************************************************************/
/* Once the INITIALIZE OR RESTART commands have been received & processed, */
/* this function serves as the command controller for the experiment. When */
/* command are received the command list is examined and the appropriate   */
/* function called.  The watchdog timer is called in the loop while        */
/* waiting for a command to preclude timing out.  A check to see if a      */
/* scheduled experiment event has occurred is done each loop and a         */
/* reading from the temperature sensors is initiated every 32 seconds at   */
/* approximately the half second mark.                                     */
/****************************************************************************/

void run_exper(void)
{

   extern unsigned char rbuf[];
   extern unsigned char cmdlist[], cmdptr, nextcmd, nextevent;
   extern DATAPKT dmp_buf;
   extern unsigned long seconds;
   extern SOHPKT soh_buf;
   extern unsigned int FATIGUE_CAPS;
   extern register char tick;

   while(TRUE)
      {
      if(cmdptr != nextcmd)
         {
         switch(cmdlist[nextcmd])
            {
            case SOH:
                        POPCMD;
                        send_soh();
                        clr_soh();
                        break;

            case DATADUMP:
                        POPCMD;
                        send_dump(&dmp_buf.head.flag);
                        clr_buf();
                        break;

            case UTIME:
                        POPCMD;
                        if (rbuf[CTRL_FLD] & ACK_RESPNS)
                           send_ack(UTIME);
                        set_utime();
                        break;

            case SHUTDOWN:
                        POPCMD;
                        send_dump(&dmp_buf.head.flag);
                        send_soh();
                        break;

            case TESTMODE:
                        POPCMD;
                        if (rbuf[CTRL_FLD] & ACK_RESPNS)
                           send_ack(TESTMODE);
                        FATIGUE_CAPS = FALSE;
                        soh_buf.status=0;
                        break;

            case ENDTESTMODE:
```

```
                        POPCMD;
                        if (rbuf[CTRL_FLD] & ACK_RESPNS)
                           send_ack(ENDTESTMODE);
                        FATIGUE_CAPS = TRUE;  /* allow fatigue to occur    */
                        soh_buf.status=1;
                        break;
            }
         }
      if(nextevent) do_event();
      wdog();
      if(((unsigned char) seconds & TEMPMASK) == 0 && tick == HALFSEC)
         get_temps();
      }
}

/****************************************************************************/
/* This function assembles the SOH packet and calls send_packet to send it.*/
/****************************************************************************/
void send_soh(void)
{
   extern unsigned char cycle, cycleday;
   extern unsigned char rbuf[];
   extern SOHPKT soh_buf;

   soh_buf.head.flag = FLAG;
   soh_buf.head.size_hi = 0x00;
   soh_buf.head.size_lo = SOHSIZE;           /* data field = 39 bytes      */
   soh_buf.head.dest_hi = rbuf[SRC_FLD];
   soh_buf.head.dest_lo = rbuf[SRC_FLD +1];
   soh_buf.head.src_hi  = NPSID_HI;          /* hi-byte addr sender of cmd */
   soh_buf.head.src_lo  = NPSID_LO;          /* lo-byte addr sender of cmd */
   soh_buf.head.ctrl_hi = 0x04;              /* ack packet                 */
   soh_buf.head.ctrl_lo = 0x00;              /* sequence byte not used     */
   soh_buf.cycle = cycle;                    /* current cycle status       */
   soh_buf.cycleday = cycleday;
   read_time();                         /* read seconds timer store in soh */
   send_packet((unsigned char *) &soh_buf);
}

/****************************************************************************/
/* This function assembles the acknowledge packet and calls send_packet to */
/* send it.                                                                */
/****************************************************************************/
void send_ack(unsigned char cmd)
{
   extern unsigned char rbuf[];
   extern ACKPKT ack_buf;

   ack_buf.head.flag = FLAG;
   ack_buf.head.size_hi = 0x00;
   ack_buf.head.size_lo = ACKSIZE;        /* size of ACK packet data fld   */
   ack_buf.head.dest_hi = rbuf[SRC_FLD];
   ack_buf.head.dest_lo = rbuf[SRC_FLD +1];
   ack_buf.head.src_hi  = NPSID_HI;    /* hi-byte  addr sender of cmnd     */
   ack_buf.head.src_lo  = NPSID_LO;    /* low-byte addr sender of cmnd     */
   ack_buf.head.ctrl_hi = 0x04;        /* responding to ack request        */
   ack_buf.head.ctrl_lo = 0x00;        /* sequence byte not used           */
   ack_buf.ack_cmd = cmd;
   send_packet((unsigned char *) &ack_buf);
}

/****************************************************************************/
/* This function assembles the data dump packet and calls send_packet to   */
/* send it.                                                                */
/****************************************************************************/
```

```
void send_dump(unsigned char *dhead)
{
   extern unsigned char rbuf[];
   extern unsigned char cycle, cycleday;

   dhead[0]            = FLAG;
   dhead[PLEN]         = DMPSIZE_HI;
   dhead[PLEN+1]       = DMPSIZE_LO;
   dhead[DEST_FLD]     = rbuf[SRC_FLD];        /* hi-byte sender of cmnd      */
   dhead[DEST_FLD+1]   = rbuf[SRC_FLD +1];     /* low_byte sender of cmnd     */
   dhead[SRC_FLD]      = NPSID_HI;             /* ferro address               */
   dhead[SRC_FLD+1]    = NPSID_LO;
   dhead[CTRL_FLD]     = 0x04;                  /* ack packet                 */
   dhead[CTRL_FLD+1]   = 0x00;                  /* sequence field not used    */
   dhead[9] = cycle;
   dhead[10] = cycleday;
   send_packet(dhead);
}

/*****************************************************************************/
/* This function executes experiment events.  Events will not be executed   */
/* while in Test Mode.  WRT_AGE1 writes to the fcaps in the 2 ageing DUTs   */
/* to start the experiment.  Subsequent writes for ageing purposes will     */
/* be taken care immediately following a read operation.  WRT_AGE1 also    */
/* starts the fatiguing of the fcaps in the fatigue DUTs. WRT_AGE2          */
/* is used to start off the ageing cycle of the fcaps in the fatigue DUTs. */
/* RD_AGE1 is used to read the fcaps in the ageing DUTs at the end of an    */
/* ageing period.  A pointer to the proper location in the data dump        */
/* buffer is calculated and passed to the fcap read function.  The          */
/* AGEREADCYCLE variables count the # of ageing reads that have occured     */
/* since the last data dump.  The polarity of both the reads and writes to */
/* the fcaps is switched each cycle.  After the fcaps have been read, new   */
/* values are written to start the next ageing period. RD_AGE2 performs     */
/* similar functions for the ageing cycle of the fatigue DUTs.  The RD_FAT */
/* event causes the fatigue DUTs to be read during the fatigue cycle.       */
/* STRT_FAT starts the fatigue operation by connecting the PWM to the       */
/* fatigue DUTs.                                                             */
/*****************************************************************************/
void do_event(void)
{
   extern unsigned char cycle;
   extern unsigned char nextevent;
   extern unsigned char porta, portb, portc;
   extern unsigned char agereadcycle1, agereadcycle2;
   extern DATAPKT dmp_buf;
   extern unsigned int FATIGUE_CAPS;
   unsigned char *ptr, i;

   if (!FATIGUE_CAPS)                 /* no events if in TEST MODE            */
      nextevent = NULLEVENT;
   wdog();

   switch(nextevent)
      {
      case WRT_AGE1: portc = (cycle & 0x01) ? 0x08 : 0; /* inv on odd cycles */
                     write_dut(AGEDUT1);          /* if flag set, fatigue  */
                     startfatigue();       /* start fatigue at beg of cycle */
                     WAIT1: asm bbs ios0, 0, WAIT1;     /* til end of pulse */
                     write_dut(AGEDUT2);
                     portc = 0;
                     break;

      case WRT_AGE2: stopfatigue();            /* stop fatigue to age 1 day */
                     portc = (cycle & 0x01) ? 0x08 : 0;  /* inv on odd cycles */
                     write_dut(FATDUT1);
                     WAIT2: asm bbs ios0, 0, WAIT2;
```

```
                     write_dut(FATDUT2);
                     portc = 0;
                     break;

      case RD_AGE1:  ptr = dmp_buf.adata + (agereadcycle1 << 4);
                     read_ageing(AGEDUT1,ptr);
                     ptr += 8;
                     read_ageing(AGEDUT2,ptr);
                     portc = (cycle & 0x01) ? 0x08 : 0;  /* inv on odd cycles */
                     write_dut(AGEDUT1);
                     WAIT3: asm bbs ios0, 0, WAIT3;       /* til end of pulse */
                     write_dut(AGEDUT2);
                     portc = 0;
                     agereadcycle1++;
                     break;

      case RD_AGE2:  ptr = dmp_buf.fdata + (agereadcycle2 << 4);
                     read_ageing(FATDUT1, ptr);
                     ptr += 8;
                     read_ageing(FATDUT2, ptr);
                     portc = (cycle & 0x01) ? 0x08 : 0;  /* inv on odd cycles */
                     write_dut(FATDUT1);
                     WAIT4: asm bbs ios0, 0, WAIT4;       /* til end of pulse */
                     write_dut(FATDUT2);
                     portc = 0;
                     agereadcycle2++;
                     break;

      case RD_FAT:   stopfatigue();
                     ptr = dmp_buf.fdata;
                     for (i=0; i < 8; i++)
                        {
                        getfatigue(FATDUT1,i,ptr);
                        ptr += 2;
                        wdog();
                        }
                     for (i=0; i < 8; i++)
                        {
                        getfatigue(FATDUT2,i,ptr);
                        ptr += 2;
                        wdog();
                        }
                     startfatigue();
                     break;

      case STRT_FAT: startfatigue();
                     break;
   };
   nextevent = NULLEVENT;
}

/*****************************************************************************/
/* This procedure waits for the command specified by the argument.  The    */
/* watchdog timer is called while in the busy loop.  If a command other     */
/* than the specified command is received, it is removed from the command  */
/* list and ignored.  When the proper command is received, the command is  */
/* removed from the list, acknowledged if necessary, and the loop is       */
/* terminated.                                                               */
/*****************************************************************************/

void wait_for_command(unsigned char command)
{
   extern unsigned char rbuf[];
   extern unsigned char cmdlist[], nextcmd, cmdptr;

   while (TRUE)
```

```
      {
      while (nextcmd == cmdptr)
         wdog();
      if (cmdlist[nextcmd] == command)
         {
         POPCMD;
         if (rbuf[CTRL_FLD] & ACK_RESPNS)
            send_ack(command);
         break;
         }
      else POPCMD;
      }
}
/****************************************************************************/
/* This function waits for the two PARAMETER packets that follow the UTIME */
/* packet.  The 2 data bytes in each packet are retrieved and stored in    */
/* the GPS variable.  GPS is a union that maps 4 unsigned char variables   */
/* onto the unsigned long containing the GPS time.  The 4 bytes making up  */
/* GPS time are converted to little endian order as required by the 80C196 */
/* Next the seconds elapsed on the current day are extracted by using the  */
/* mod operator.  A critical region is used when the experiment clock is   */
/* modified.                                                               */
/****************************************************************************/

void set_utime(void)
{
   extern unsigned char rbuf[];
   extern TIME GPS;
   extern unsigned long seconds;

   wait_for_command(PARAMETER);
   GPS.buf[3]=rbuf[HEAD_LEN + 1];      /* 2 MSB's */
   GPS.buf[2]=rbuf[HEAD_LEN + 2];
   wait_for_command(PARAMETER);
   GPS.buf[1]=rbuf[HEAD_LEN + 1];      /* 2 LSB's */
   GPS.buf[0]=rbuf[HEAD_LEN + 2];

   GPS.secs = GPS.secs % ONEDAY;       /* just need the seconds since today */

   asm di;                             /* critical region */
   seconds = GPS.secs;                 /* set the clock */
   asm ei;
}

/****************************************************************************/
/* This procedure reads the value from the experiment clock.  The # of     */
/* seconds is converted from little endian to big endian and stored in the */
/* SOH packet at the appropriate location.                                 */
/****************************************************************************/

void read_time(void)
   {
   extern  SOHPKT soh_buf;
   extern unsigned long seconds ;
   TIME sohtime;

   asm   di;                           /* critical region */
   sohtime.secs = seconds;
   asm   ei;

   soh_buf.time[3] = sohtime.buf[0];   /* convert from little to big endian */
   soh_buf.time[2] = sohtime.buf[1];
   soh_buf.time[1] = sohtime.buf[2];
   soh_buf.time[0] = sohtime.buf[3];

}
```

```
                          /* EXPER.C */

#include "ferro.h"

/****************************************************************************/
/* This function initializes the PWM and routes the pulses to the fatigue  */
/* DUTs.                                                                    */
/****************************************************************************/

void startfatigue(void)
{
   extern unsigned char portb;

   /* initialize PWM */
   pwm_control = 128;            /* setup for 50% duty cycle */
   ioc1 |= 0x01;                 /* select PWM */
   portb = 0xA5;                 /* connect FATDUT1 & FATDUT2  to PWM */
}

/****************************************************************************/
/* This function disables the PWM function and applies ground to the       */
/* input of the fatigue DUTs                                                */
/****************************************************************************/

void stopfatigue(void)
{
   extern unsigned char portb;

   ioc1 &= 0xFE;              /* deselect PWM function */
   portb = 0xF0;              /* ground input to duts */
}

/****************************************************************************/
/* This function schedules a write pulse to be generated in the HSO CAM.   */
/* Timer 1 is used for the time base and is reset to start the operation.  */
/****************************************************************************/

void write_pulse(void)
{
   wsr = 0x0F;                /* select alternate reg window functions */
   asm di;                    /* disable interrupts until CAM loaded   */
   timer1 = 0x0000;                   /* reset TIMER1 */
   wsr = 0x00;
   hso_command = 0x20;               /* set HSO.0 high */
   hso_time = STARTHI;                /* start pulse after delay */
   WAIT_FOR_CAM;
   hso_command = 0x00;                /* set HSO.0 low */
   hso_time = ENDHI;
   asm ei;                    /* restore interrupts */
}

/****************************************************************************/
/* This function schedules a read pulse to be generated in the HSO CAM.    */
/* Timer 1 is used for the time base and is reset to start the operation.  */
/* An A/D conversion is scheduled to occur 15 counts before the end of the */
/* pulse so that the pulse will remain high during the entire conversion.  */
/* Once the pulse is scheduled, a loop is executed until the A/D conv is   */
/* complete.                                                                */
/****************************************************************************/

void read_pulse(void)
{
   wsr = 0x0F;                /* select alternate reg window functions */
   asm di;                    /* disable interrupts til CAM loaded */
   timer1 = 0x0000;                   /* reset TIMER1 */
   wsr = 0x00;
```

```
   hso_command = 0x20;                       /* set HSO.0 high */
   hso_time = STARTHI;                        /* start pulse after delay */
   WAIT_FOR_CAM;
   hso_command = 0x0F;                        /* start A/D conversion */
   hso_time = AD_TIME;
   WAIT_FOR_CAM;
   hso_command = 0x00;                        /* set HSO.0 low */
   hso_time = ENDHI;
   asm ei;                    /* reenable interrupts */
   WAIT1: asm bbc int_pending, AD_COMP, WAIT1;  /* Wait for A/D complete */
   asm andb int_pending, #CLR_AD_COMP;
}

/****************************************************************************/
/* This function schedules a read pulse to be generated in the HSO CAM.    */
/* Timer 1 is used for the time base and is reset to start the operation.  */
/* An A/D conversion is scheduled to occur 75 counts after the beginning   */
/* of the pulse to allow for the circuit to stabilize.  Eight A/D          */
/* conversions occur during the pulse. Only the first conversion is        */
/* scheduled using the HSO; the remaining conversions are carried out in   */
/* succession following the first.                                          */
/****************************************************************************/

void age_read_pulse(void)
{
   wsr = 0x0F;                /* select alternate reg window functions */
   timer1 = 0x0000;                   /* reset TIMER1 */
   wsr = 0x00;
   hso_command = 0x20;               /* set HSO.0 high */
   hso_time = STARTHI;                /* start pulse after delay */
   WAIT_FOR_CAM;
   hso_command = 0x0F;                /* start A/D conversion */
   hso_time = FIRST_AD;
   WAIT_FOR_CAM;
   hso_command = 0x00;                /* set HSO.0 low */
   hso_time = ENDHI;
}

/****************************************************************************/
/* This function initiate A/D conversions to read the 4 temperature        */
/* sensors.  A wait loop spins until the conversion is complete and the    */
/* 10-bit result is rounded with the most significant 8 bits stored in the */
/* SOH buffer.                                                              */
/****************************************************************************/

void get_temps(void)
{
   extern SOHPKT soh_buf;
   unsigned char i;

   for (i=0; i < 4; i++)
      {
      ad_command = 0x0C + i;
      WAIT1: asm bbc int_pending, AD_COMP, WAIT1;  /* Wait for A/D complete */
      asm andb int_pending, #CLR_AD_COMP;
      soh_buf.temp[i] = ad_result_hi + ROUND;
      }
}

/****************************************************************************/
/* This function reads the 8 fcaps in the DUT specified by the first       */
/* argument.  The polarity of the read pulse is alternated each experiment */
/* cycle.  The pulse and first A/D conversion are scheduled in the HSO by  */
/* calling age_read_pulse.  The remaining conversions are carried out in   */
/* succession following the first.  The results of the conversions are    */
/* rounded and the most significant 8 bits stored in the data buffer at    */
/* the location pointed to by the second argument, bufptr.                 */
/* Interrupts must be disabled during the read pulse to ensure all 8 A/D   */
/* conversions complete before the end of the pulse.                       */
/***************************************************************************/

void read_ageing(unsigned char dut_sel, unsigned char *bufptr)
{
   extern unsigned char porta, portb, portc;       /* 8255 ports */
   extern unsigned char cycle;
   extern const unsigned char rd_dut_val[];

   unsigned char i;
   register unsigned char invert;
   register unsigned char tempb;

   porta = rd_dut_val[dut_sel];
   tempb = portb;                                /* save portb settings */
   portb = tempb | (0x11 << dut_sel);            /* disconnect gnd on dut */
   invert = (cycle & INV) ? 0 : 0x08;            /* negative pls on odd cycle */
   portc = invert;

   ad_command = 0x00;                        /* A/D channel 0, wait for HSO */
   asm di;                                   /* disable interrupts during pulse */
   age_read_pulse();
   for(i=0; i < 8; )
      {
      WAIT1: asm bbc int_pending, AD_COMP, WAIT1;  /* Wait for A/D complete */
      portc = ++i + invert;
      asm andb int_pending, #CLR_AD_COMP;
      bufptr[i] = ad_result_hi + ROUND;
      ad_command = 0x08;                           /* start A/D */
      }
   asm ei;

   porta = 0;                                      /* disable selects */
   portb = tempb;                                  /* restore portb setting */
   portc = 0;
}

/***************************************************************************/
/* This function performs 2 write-then-read combinations of opposite       */
/* polarity for the fcap specified by the dut and fcap arguments.  The     */
/* results of the A/D conversions are stored in the data buffer at the     */
/* location pointed to by the argument bufptr.  The functions write_dut    */
/* and read_pulse are called to generate the pulses and schedule the A/D   */
/* conversions for the write and read operations.                          */
/***************************************************************************/

void getfatigue(unsigned char dut,unsigned char fcap,unsigned char * bufptr)
{
   extern unsigned char porta, portb, portc;       /* 8255 ports */
   extern const unsigned char wrt_dut_val[];
   extern const unsigned char rd_dut_val[];

   register unsigned char tempb;

   portc = 0x08;                              /* Send negative write pulse */
   write_dut(dut);

   porta = rd_dut_val[dut];
   tempb = portb;                               /* save portb setting */
   portb = tempb | (0x11 << dut);               /* disconnect gnd on dut */
   portc = fcap;                                /* positive read of fcap */

   ad_command = 0x00;                       /* A/D channel 0, wait for HSO */
   read_pulse();
```

```
   bufptr[0] = ad_result_hi + ROUND;

   portc = 0;                               /* Send positive write pulse */
   write_dut(dut);

   porta = rd_dut_val[dut];
   portb = tempb | (0x11 << dut);                 /* disconnect gnd on dut */
   portc = fcap + 0x18;                    /* Same channel do negative part */

   ad_command = 0x00;                     /* A/D channel 0, wait for HSO */
   read_pulse();
   bufptr[1] = ad_result_hi + ROUND;

   porta = 0;                                        /* disable selects */
   portb = tempb;
   portc = 0;
}

/***************************************************************************/
/* This function writes to the DUT specified in the argument.  The         */
/* 8255 ports are set to route the pulse to the selected DUT and the       */
/* write_dut function is called to generate the write pulse.               */
/***************************************************************************/

void write_dut(unsigned char dut_sel)
{
   extern unsigned char porta, portb;              /* 8255 ports */
   register unsigned char tempb;

   porta = wrt_dut_val[dut_sel];                   /* set routing switch */
   tempb = portb;
   portb = tempb | (0x11 << dut_sel);              /* disconnect gnd on dut */
   write_pulse();
   WAIT1: asm bbs ios0, 0, WAIT1;                  /* wait til end of pulse */
   porta = 0;                                      /* disable selects */
   portb = tempb;
}
```

```
                          /* PACKET.C */
#include "ferro.h"

/* points serial receive interrupt vector to rcv_packet function */
#pragma interrupt(rcv_packet=25)

/*****************************************************************************/
/* This function serves the serial port receive interrupt and assembles     */
/* the packet as it is received.  The bytes of each packet are received     */
/* one at a time.  The packet must start with a FLAG byte; incoming chars   */
/* are ignored until a FLAG is received.  The next 2 bytes are the packet   */
/* length field.  Once the length is received, the end of the packet can    */
/* be calculated.  When all bytes of the packet are received, the checksum */
/* is validated and the source ID is checked.  If the packet is valid, the */
/* command code from the packet command field is placed in the command      */
/* list.  If the FLAG character occurs as data within the packet, the       */
/* transmitter will insert an extra FLAG character in the data stream.      */
/* The state variable is used to look for the double FLAG so the extra      */
/* FLAG can be stripped.  If a single FLAG occurs in the body of the        */
/* packet, an error has occurred and the packet is ignored.                 */
/*****************************************************************************/

void rcv_packet(void)
{
   extern register unsigned char eobuf, rcv_ptr, state;
   extern unsigned char rbuf[],cmdlist[], cmdptr;
   register unsigned char ch;
   unsigned int len, temp;

   if (sp_stat & 0x14 )
      {
      rcv_ptr = 0;
      return;
      }
   ch = sbuf;
   switch (rcv_ptr)
      {
      case 0: if (ch == FLAG)
                  rbuf[rcv_ptr++] = ch ;
              break ;
      case 1:  if(ch == FLAG)
                  {
                  if(state = !(state))
                     break;
                  }
               else
                  if(state)
                     state = FALSE;            /* single FLAG so resync */
               rbuf[rcv_ptr++] = ch;        /* not FLAG or double FLAG */
               break;
      case 2:  if(ch == FLAG)
                  {
                  if(state = !(state))
                     break;
                  }
               else
                  if(state)
                     {
                     rcv_ptr = 1;
                     state = FALSE;            /* single FLAG so resync */
                     rbuf[rcv_ptr++] = ch;     /* must be new 2nd byte  */
                     break;
                     }
               rbuf[rcv_ptr++] = ch;        /* not FLAG or double FLAG */
               eobuf = ch + OVERHEAD;
               break;
```

```
     default: if(ch == FLAG)
                  {
                  if(state = !(state))
                     break;
                  }
               else
                  if(state)
                     {
                     rcv_ptr = 1;              /* single FLAG so resync */
                     state = FALSE;
                     }
               rbuf[rcv_ptr++] = ch;        /* not FLAG or double FLAG */
               if(rcv_ptr == eobuf)
                  {
                  rcv_ptr = 0;
                  temp = rbuf[PLEN];
                  len = (temp <<8) + rbuf[PLEN + 1] + OVERHEAD;
                  if(rbuf[DEST_FLD] == NPSID_HI && rbuf[DEST_FLD+1] ==
                     NPSID_LO && valid_csum(rbuf, len))
                     {
                     cmdlist[cmdptr] = rbuf[HEAD_LEN];   /* store command */
                     cmdptr = (cmdptr + 1) & 0x03;       /* update pointer */
                     }
                  }
     }

}

/*****************************************************************************/
/* This function transmits the packet pointed to by the argument.  The      */
/* length of the packet is calculated and a checksum for the packet is      */
/* calculated.  If a FLAG character occurs in the data stream after the     */
/* first byte, an extra FLAG is inserted as an escape sequence.  For this   */
/* case the double buffering feature of the serial port is used.            */
/*****************************************************************************/
void send_packet(unsigned char *buf)
{
   unsigned i,temp, len;

   temp = buf[PLEN];     /* calculate length of packet */
   len = (temp << 8) + buf[PLEN + 1] + OVERHEAD;

   compute_checksum(buf);
   while(!(sp_stat & TXE));                      /* wait for empty buffer */
   sbuf = buf[0];                                /* begin packet */
   for(i=1; i < len; i++)                        /* change to send dumps */
     {
     wdog();
     while(!(sp_stat & TXE));                    /* wait for empty buffer */
     sbuf = buf[i];
     if(buf[i] == FLAG)
        sbuf = FLAG;                             /* transmit double flag */
     }
}
```

```
                               /* CHECK.C */
#include "ferro.h"

/**************************************************************************/
/* This function computes the checksum of the packet pointed to by the     */
/* argument.  The 2-byte checksum is appended to the end of the packet.    */
/* The length of packet is calculated from the data length field of the    */
/* packet.                                                                 */
/**************************************************************************/

void compute_checksum(unsigned char *buf)
{
   int len, i, temp;
   register unsigned char c0 = 0, c1 = 0;

   temp = buf[PLEN];
   len = (temp << 8)  +  buf[PLEN + 1] + HEAD_LEN;
   buf[len] = 0;
   buf[len + 1] = 0;
   for(i = 0; i < len +2; i++)
      {
      c0 += buf[i];
      c1 += c0;
      }
   buf[len] = c0 - c1;
   buf[len + 1] = c1 - (c0 << 1);
}

/**************************************************************************/
/* This function validates the checksum of the buffer pointed to by the    */
/* first argument.  The length of the buffer is specified by the second    */
/* argument, len.  The length was passed as a parameter rather than        */
/* calculated from the packet length field to allow buffers other than     */
/* packets to be validated.  If the checksum is valid, the function        */
/* returns TRUE.                                                           */
/**************************************************************************/

int valid_csum(unsigned char *packet, unsigned int len)
{
   int i;
   register unsigned char c0 = 0, c1 = 0;

   for(i = 0; i < len; i++)
      {
      c0 += packet[i];
      c1 += c0;
      }
   return((c0 == 0) && (c1 == 0));
}
```

APPENDIX C -- TEST/VERIFICATION SOFTWARE SOURCE CODE

The test and verification software consists of two header files and three source files. SERIAL.H contains all of the definitions and declarations necessary to initialize and use the RS-232 serial port on an IBM PC compatible computer. The routines for initializing and using the port for communication are located in SERIAL.C.

TERM.H contains the definitions and declarations for the main portion of the test and verification software. Header files for the Ultra Windows library are included to provide definitions and declarations to support the library functions used to implement the user interface. Constants used to define the FERRO commands and packet structure are also defined here.

TERM.C contains the routines that control and interact with the user interface. The routines to display menus and windows, and implement menu selections are also located in this file. The routines to send, receive and display packets are located in the file T_PACKET.C.

```
/****************************** SERIAL.H ******************************/

#define TIME_OUT  6
#define ON        1
#define OFF       0
#define TRUE      1
#define FALSE     0
#define SBUFSIZ   0x4000
#define SPEEDNUM  115200L
#define BUFOVFL   1

#define COM1          1
#define COM2          2
#define COM1BASE      0x3F8   /*  Base port address for COM1  */
#define COM2BASE      0x2F8   /*  Base port address for COM2  */

/*********************************************************************/
/*  The 8250 UART has 10 registers accessible through 7 port addresses. */
/*  Here are their addresses relative to COM1BASE and COM2BASE. Note    */
/*  that the baud rate registers, (DLL) and (DLH) are active only when  */
/*  the Divisor-Latch Access-Bit (DLAB) is on. The (DLAB) is bit 7 of   */
/*  the (LCR).                                                          */

/*********************************************************************

    -  TXR Output data to the serial port.
    -  RXR Input data from the serial port.
    -  LCR Initialize the serial port.
    -  IER Controls interrupt generation.
    -  IIR Identifies interrupts.
    -  MCR Send contorl signals to the modem.
    -  LSR Monitor the status of the serial port.
    -  MSR Receive status of the modem.
    -  DLL Low byte of baud rate divisor.
    -  DHH High byte of baud rate divisor.

*********************************************************************/
#define TXR           0       /*  Transmit register (WRITE) */
#define RXR           0       /*  Receive register  (READ)  */
#define IER           1       /*  Interrupt Enable          */
#define IIR           2       /*  Interrupt ID              */
#define LCR           3       /*  Line control              */
#define MCR           4       /*  Modem control             */
#define LSR           5       /*  Line Status               */
#define MSR           6       /*  Modem Status              */
#define DLL           0       /*  Divisor Latch Low         */
#define DLH           1       /*  Divisor latch High        */


/*******************************************************************/
  Bit values held in the Line Control Register (LCR).
    bit               meaning
    ---               -------
    0-1               00=5 bits, 01=6 bits, 10=7 bits, 11=8 bits.
    2                 Stop bits.
    3                 0=parity off, 1=parity on.
    4                 0=parity odd, 1=parity even.
    5                 Sticky parity.
    6                 Set break.
    7                 Toggle port addresses.
/*******************************************************************/

#define NO_PARITY     0x00
#define EVEN_PARITY   0x18
#define ODD_PARITY    0x08
```

```
/*********************************************************************/
  Bit values held in the Line Status Register (LSR).
    bit                meaning
    ---                -------
     0                 Data ready.
     1                 Overrun error - Data register overwritten.
     2                 Parity error - bad transmission.
     3                 Framing error - No stop bit was found.
     4                 Break detect - End to transmission requested.
     5                 Transmitter holding register is empty.
     6                 Transmitter shift register is empty.
     7                 Time out - off line.

/*********************************************************************/

#define RCVRDY        0x01
#define OVRERR        0x02
#define PRTYERR       0x04
#define FRMERR        0x08
#define BRKERR        0x10
#define XMTRDY        0x20
#define XMTRSR        0x40
#define TIMEOUT       0x80

/*********************************************************************/
  Bit values held in the Modem Output Control Register (MCR).
    bit               meaning
    ---               -------
     0                Data Terminal Ready. Computer ready to go.
     1                Request To Send. Computer wants to send data.
     2                auxillary output #1.
     3                auxillary output #2.(Note: This bit must be
                      set to allow the communications card to send
                      interrupts to the system)
     4                UART ouput looped back as input.
    5-7               not used.
/********************************************************************/

#define DTR           0x01
#define RTS           0x02
#define MC_INT        0x08

/********************************************************************/
  Bit values held in the Modem Input Status Register (MSR).
        bit                meaning
        ---                -------
         0                 delta Clear To Send.
         1                 delta Data Set Ready.
         2                 delta Ring Indicator.
         3                 delta Data Carrier Detect.
         4                 Clear To Send.
         5                 Data Set Ready.
         6                 Ring Indicator.
         7                 Data Carrier Detect.

/********************************************************************/

#define CTS           0x10
#define DSR           0x20

/********************************************************************/
  Bit values held in the Interrupt Enable Register (IER).
        bit            meaning
        ---            -------
         0             Interrupt when data received.
         1             Interrupt when transmitter holding reg. empty.
        2               Interrupt when data reception error.
        3               Interrupt when change in modem status register.
        4-7             Not used.
/*********************************************************************/

#define RX_INT          0x01

/*********************************************************************/
  Bit values held in the Interrupt Identification Register (IIR).
        bit                meaning
        ---                -------
        0               Interrupt pending
        1-2             Interrupt ID code
                        00=Change in modem status register,
                        01=Transmitter holding register empty,
                        10=Data received,
                        11=reception error, or break encountered.
        3-7             Not used.
/*********************************************************************/
#define RX_ID           0x04
#define RX_MASK         0x07


/*
    These are the port addresses of the 8259 Programmable Interrupt
    Controller (PIC).
*/
#define IMR             0x21   /* Interrupt Mask Register port */
#define ICR             0x20   /* Interrupt Control Port       */


/*
    An end of interrupt needs to be sent to the Control Port of
    the 8259 when a hardware interrupt ends.
*/
#define EOI             0x20   /* End Of Interrupt */


/*
    The (IMR) tells the (PIC) to service an interrupt only if it
    is not masked (FALSE).
*/
#define IRQ3            0xF7  /* COM2 & COM4 */
#define IRQ4            0xEF  /* COM1 & COM3 */


/*
    The (IMR) tells the (PIC) to service an interrupt only if it
    is not masked (FALSE).
*/
#define IRQ3            0xF7  /* COM2 & COM4 */
#define IRQ4            0xEF  /* COM1 & COM3 */

/* Structure to contain serial port parameters */

struct param {
             int port;                  /* COM 1 or 2 */
             int speed;                 /* Baud rate */
             int parity;                /* Even Odd or None */
             int bits;                  /* # Data bits */
             int stopbit;               /* # Stop bits */
             int status;                /* Port on or off (Boolean) */
             char logname[13];          /* Name of log file */
             int logstat;               /* Log on or off (Boolean) */
             };
```

```
/********************** Function declarations *****************************/
void interrupt com_int (void);
int getccb(void);
void closeserial(void);
void initserial (void);
void i_disable (void);
void i_enable (int pnum);
void comon (void);
int serial_out (char x);
int setserial (struct param *serial);
```

```
/********************************** SERIAL.C *********************************/
#include        "serial.h"
#include        <dos.h>
#include        <stdio.h>
#include        <conio.h>
#include        <bios.h>
#include        <process.h>
#include        <time.h>

int SError;
int portbase=0;
void interrupt (*oldvects[2])();
char ccbuf[SBUFSIZ];
int startbuf=0;
int endbuf=0;

struct param serial = {COM1, 2400, NO_PARITY, 8, 1, 0, "FERRO.LOG", 0};

/*************************************************************************/
/* This is the serial interrupt service routine.  Only the receive cycle */
/* is service by the interrupt. The character is read from the serial    */
/* port and placed in a circular buffer.                                 */
/*************************************************************************/

void interrupt com_int (void)

{
   disable ();
   if ((inportb (portbase+IIR) & RX_MASK)==RX_ID)
      {
      if (((endbuf+1) & SBUFSIZE - 1)==startbuf)
         SError=BUFOVFL;
      ccbuf[endbuf++] = inportb (portbase+RXR);
      endbuf &= SBUFSIZE-1;
      };
   outportb (ICR,EOI);
   enable ();
}

/* this routine returns the next character in the serial buffer */

int getc_buf(void)

{
   int res;

   if (endbuf==startbuf) return (-1);
   res=(int) ccbuf[startbuf];
   startbuf=(startbuf+1) % SBUFSIZ;
   return (res);
}

/* set up the serial port to interrupt and enable interrupts          */

void i_enable (int pnum)

{
   int c;
   disable();
   c = inportb (portbase+MCR) | MC_INT;
   outportb (portbase+MCR,c);
   outportb (portbase+IER,RX_INT);
   c = inportb (IMR) & (pnum==2 ? IRQ3 : IRQ4);
   outportb (IMR,c);
   enable();
}
```

```
/* disable serial port interrupts                                      */

void i_disable (void)

{
   int c;
   disable ();
   c=inportb (IMR) | -IRQ3 | -IRQ4;
   outportb (IMR,c);
   outportb (portbase + IER,0);
   c=inportb (portbase+MCR) & ~MC_INT;
   outportb (portbase+MCR,c);
   enable ();
}

/* Turn on the comm port                                              */

void comon (void)

{
   int c, pnum;

   pnum = portbase == COM1BASE ? 1 : 2;
   i_enable (pnum);
   c=inportb (portbase + MCR) | DTR | RTS;
   outportb (portbase + MCR,c);
}

/* Initialize the buffer pointers, save the old serial interrupt vectors */
/* and set the new ones                                                   */

void initserial (void)

{
   endbuf=startbuf=0;                   /* start with empty buffer */
   oldvects[0]=getvect(0x0b);           /* save old serial int vector */
   oldvects[1]=getvect(0x0c);
   setvect(0x0b,com_int);               /* point vector to com_int funct */
   setvect(0x0c,com_int);
   comon();                             /* turn on com port */
   serial.status = ON;
};

/* Disable the serial interrupts and restore the old serial interrupt    */
/* vectors.                                                               */

void closeserial(void)

{
   i_disable ();                        /* disable serial port interrupt */
   outportb (portbase+MCR,0);           /* reset MCR to zero */
   setvect(0x0b,oldvects[0]);           /* restore serial interrupt vector */
   setvect(0x0c,oldvects[1]);
   serial.status = OFF;
};

/* this outputs a serial character              */

int serial_out(char x)

{
   time_t btime;

   outportb (portbase+MCR,MC_INT|DTR|RTS);
   /* wait for clear to send     */

   while ((inportb(portbase + MSR) & CTS)==0);


   /* wait for outport register to clear          */
   while ((inportb(portbase+LSR) & DSR)==0);
   disable ();
   outportb (portbase+TXR,x);
   enable ();
   return (0);
}

/* this routine sets which port we are working with     */

int setport (int Port)

{
   int far * RS232_Addr;
   int Offset;

   switch (Port)                        /* sort out the base address    */
      {
      case 1 : Offset=0x0000; break;    /* only ports one & two allowed */
      case 2 : Offset=0x0002; break;
      default : return (-1);
      };
   RS232_Addr=MK_FP(0x0040,Offset);     /* find out where the port is   */
   if (*RS232_Addr==0) return (-1);     /* if it ain't there return (-1)*/
   portbase=*RS232_Addr;                /* otherwise set portbase       */
   return (0);
}

/* this routine sets the baud rate register to the correct count for
   Speed baud  */

int setspeed (int Speed)

{
   char c;
   int divisor;

   if (Speed==0) return (-1);                   /* avoid divide by zero */
   else divisor=(int)(SPEEDNUM/Speed);
   if (portbase==0) return (-1);
   disable ();                                         /* disable ints */
   c=inportb (portbase+LCR);
   outportb (portbase+LCR,(c|0x80));                   /* set DLAB in LCR */
   outportb (portbase+DLL,(divisor & 0x00ff));         /* set divisor  */
   outportb (portbase+DLH,((divisor>>8)&0x00ff));
   outportb (portbase+LCR,c);                          /* restore LCR */
   enable();                                           /* reenable ints */
   return (0);
}

/*  This routine sets up the LCR for the selected parity, number of
    data bits and number of stop bits */

int setLCR (int Parity,int Bits,int StopBit)

{
   int temp;
   if (portbase==0) return (-1);
   if ((Parity<NO_PARITY) || (Parity>ODD_PARITY)) return (-1);
   if ((Bits<5) || (Bits>8)) return (-1);
   if ((StopBit<1) || (StopBit>2)) return (-1);
```

```
   temp=Bits-5;
   temp|=((StopBit==1) ? 0x00 : 0x04);
   switch (Parity)
      {
      case NO_PARITY : temp |= 0x00; break;
      case ODD_PARITY : temp |= 0x08; break;
      case EVEN_PARITY : temp |= 0x18; break;
      }
   disable();                          /* turn off interrupts */
   outportb (portbase+LCR,temp);
   enable();                           /* restore interrupts */
   return (0);
}

/* This routine sets the up the port by calling the appropriate routines */

int set_serial (struct param *serial)

{
   if (setport(serial->port)==-1) return(-1);
   if (setspeed(serial->speed)==-1) return(-1);
   if (setLCR(serial->parity,serial->bits,serial->stopbit)==-1) return (-1);
   return (0);
}
```

```
#include <stdlib.h>
#include <dos.h>
#include <stdio.h>
#include <alloc.h>
#include "serial.h"
#include "uw.h"
#include "uw_globx.h"
#include "uw_keys.h"


#define MAX_PACKET      1000            /* Length of maximum size packet */
#define FLAG            0x7E            /* Packet begin flag */
#define DEST_IDHI       0x4D            /* High byte, id for FERRO */
#define DEST_IDLO       0x01            /* Low byte, id for FERRO */
#define HEAD_LEN        9               /* Length of packet header */
#define OVERHEAD        HEAD_LEN+2      /* Header plus checksum */
#define PLEN            1               /* Index of length field of packet */
#define ERROR          -1
#define OK             -2

#define CONFIG        1                 /* Menu return values */
#define MESG          2
#define QUIT          3
#define PARAM         4
#define PORT          5
#define SPEED         6
#define PARITY        7
#define BITS          8
#define STOP_BIT      9
#define SV_CFG        10
#define INIT_PRT      11
#define SEND          12
#define RST_PRT       13
#define LOGFILE       14
#define LOGMSG        15
#define DONE          0
#define INIT          1                 /* values for ferro exp command */
#define DUMP          2
#define SOH           3
#define TESTMODE      4
#define ENDTESTMODE   5
#define SDOWN         6
#define RESTART       7
#define UTIME         8
#define PARAMETER     9

#define BACK_ATT  (LIGHTGRAY << 4) | BLACK      /* Menu display attributes */
#define BDR_ATT   (LIGHTGRAY << 4) | BLACK
#define CSR_ATT   (BLACK << 4) | WHITE
#define FIRST_ATT (BLACK << 4) | WHITE

/************************** Function Prototypes **********************/
void make_desk(WINDOW *wnptr);
void menu_init(void);
int  menu(void);
void menu_close(void);
void set_port(struct param *s);
void set_speed(struct param *s);
void set_parity(struct param *s);
void set_bits(struct param *s);
void set_stop(struct param *s);
void mn_close(MENU *mn_ptr);
void disp_status(struct param *s);
void comm_menu(struct param *s);
void compute_checksum(unsigned char *packet);
int  validate_checksum(unsigned char *packet, int len);
```

```
int  get_config(struct param *s);
int  save_config(struct param *s);
void put_prompt(char *prompt);
void wait_prompt(char *prompt);
void mesg_menu(void);
void add_send_wn(void);
void add_rcv_wn(void);
void del_wn(WINDOW *wnptr);
void get_time(unsigned char *utime);
int  get_days(unsigned char *days);
void get_param(unsigned char *params);
void get_logname(struct param *s);
int  send_packet(unsigned char *pkt, unsigned char *msg, unsigned char size,unsig
ned int id);
int  rcv_packet(unsigned char *msg, int *packetlen);
void disp_packet(WINDOW *wn_ptr, unsigned char *pk, unsigned int data_len);
void disp_error_packet(WINDOW *wn_ptr, unsigned char *pk, int data_len);
int  get_packet_byte(unsigned char *ch_ptr);
```

```
#include "term1.h"
/********************************** GLOBALS ***********************************/
MENU  root_menu, *root_mnp = &root_menu;
MENU  config_mn, mesg_mn, quit_mn;
MENU  *drop_mnps[3] = {&config_mn, &mesg_mn, &quit_mn};
WINDOW status_wn, *status_ptr = &status_wn;
WINDOW prompt_wn, *prompt_ptr = &prompt_wn;
WINDOW temp_wn, *temp_ptr = &temp_wn;
WINDOW send_wn, *send_ptr = &send_wn;
WINDOW rcv_wn, *rcv_ptr = &rcv_wn;
FILE *lfp;


void main()
{
   WINDOW desktop, *deskptr = &desktop;
   extern struct param serial;

   init_video(80,25);
   init_mouse();
   make_desk(deskptr);
   get_config(&serial);
   disp_status(&serial);
   menu_init();
   while (menu() != QUIT);
   if(serial.status == ON)
      closeserial();
   fclose(lfp);
   menu_close();
   wn_destroy(status_ptr);
   wn_destroy(deskptr);
   end_mouse();
   end_video();
}

/*****************************************************************************/
/* This routine sets up the desktop, creating the full screen main window   */
/* and the status window at the bottom of the screen.                       */
/*****************************************************************************/

void make_desk(WINDOW *wnptr)
{
   wn_create(0,0,79,24,DBL_BDR, WN_NORMAL, wnptr);
   wnptr->scroll = 0;                         /* turn off scrolling */
   wn_color(LIGHTGRAY, BLUE, wnptr);
   wn_bdr_color(LIGHTGRAY, BLUE, wnptr);
   wn_name("FERRO Communication Program",wnptr);
   wn_csr_pos(0, 0, wnptr);
   add_window(wnptr);
   wn_qch(1872, 177, wnptr);                  /* desktop background */

   /* Make status window */
   wn_create(1, 24, 78, 24, NO_BDR, WN_NORMAL, status_ptr);
   wn_color(BLUE, LIGHTGRAY, status_ptr);
   add_window(status_ptr);
   wn_plst(0, 0, "Status:", status_ptr);
}
```

```
/******************************************************************************/
/* This routine creates the pulldown menus and calls the Ultra Windows  w    */
/* menu handling routine.                                                     */
/******************************************************************************/

void menu_init(void)
{
/*------------------------- Create Root Menu ---------------------------*/
   menu_create(1, 1, 78, 1, M_HORIZONTAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, NO_BDR, WN_NORMAL, root_mnp);
   item_add("   Config   ", CONFIG, 3, root_mnp);
   item_add("  Messages  ", MESG, 2, root_mnp);
   item_add("    Quit    ", QUIT, 4, root_mnp);

/*------------------------- Create Config Menu -------------------------*/
   menu_create(1, 2, 19, 6, M_VERTICAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, SGL_BDR, WN_NORMAL, drop_mnps[0]);
   item_add(" Comm Parameters ", PARAM, 1, drop_mnps[0]);
   item_add(" Save Config     ", SV_CFG, 1, drop_mnps[0]);
   item_add(" Log filenam     ", LOGFILE, 1, drop_mnps[0]);

/*------------------------- Create Message Menu ------------------------*/
   menu_create(13, 2, 34, 7, M_VERTICAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, SGL_BDR, WN_NORMAL, drop_mnps[1]);
   item_add(" Initialize   Port ", INIT_PRT, 1, drop_mnps[1]);
   item_add(" Send Command       ", SEND, 1, drop_mnps[1]);
   item_add(" Log Messages       ", LOGMSG, 1, drop_mnps[1]);
   item_add(" Close Port         ", RST_PRT, 1, drop_mnps[1]);

/*------------------------- Create Quit Menu ---------------------------*/
   menu_create(25, 2, 41, 2, M_VERTICAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, SGL_BDR, WN_NORMAL, drop_mnps[2]);
   item_add("Quit      Alt-Q", QUIT, 0, drop_mnps[2]);

   menu_set(root_mnp);
}
```

```
/****************************************************************************/
/* This routine displays the pulldown menus and processes the response      */
/* returned by the Ultra Windows library menu handling routine.             */
/****************************************************************************/

int menu(void)
{
   int id;
   extern struct param serial;

   m_show();
   id = menu_system(root_mnp, drop_mnps, 0);
   switch(id)
      {
      case PARAM:
               comm_menu(&serial);
               break;
      case SV_CFG:
               save_config(&serial);
               break;
      case LOGFILE:
               get_logname(&serial);
               break;
      case INIT_PRT:
               if (set_serial(&serial))
                  wait_prompt("Unable to initialize serial port.");
               else
                  if(serial.status != ON)
                     {
                     initserial();
                     init_mouse();
                     }
               break;
      case SEND:
               mesg_menu();
               break;
      case LOGMSG:   if(serial.logstat = !serial.logstat)
                        lfp = fopen(serial.logname,"at+");
                     else
                        fclose(lfp);
                     break;
      case RST_PRT:
               if(serial.status == ON)
                  {
                  closeserial();
                  init_mouse();
                  }
               break;
      }
   disp_status(&serial);
   return(id);
}
```

```
/****************************************************************************/
/* This routine hides the restores the screen underneath the menus, hides   */
/* the menu, and releases the memory structures for the menus.              */
/****************************************************************************/

void menu_close(void)
{
   menu_restore(root_mnp);
   m_hide();
   menu_destroy(root_mnp);
   menu_destroy(drop_mnps[0]);
   menu_destroy(drop_mnps[1]);
   menu_destroy(drop_mnps[2]);
}

/****************************************************************************/
/* This routine initializes and decodes responses for the communication     */
/* parameter pulldown menu.                                                 */
/****************************************************************************/

void comm_menu(struct param *serial_ptr)
{
   int id;
   MENU comm_mn, *comm_mn_ptr = &comm_mn;

   menu_create(35, 8, 45, 15, M_VERTICAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, SGL_BDR, WN_NORMAL, comm_mn_ptr);
   item_add("Port      ", PORT, 0, comm_mn_ptr);
   item_add("Speed     ", SPEED, 0, comm_mn_ptr);
   item_add("paRity    ", PARITY, 2, comm_mn_ptr);
   item_add("Bits      ", BITS, 0, comm_mn_ptr);
   item_add("Stop bits ", STOP_BIT, 0, comm_mn_ptr);
   item_add("Done      ", DONE, 0, comm_mn_ptr);

   menu_set(comm_mn_ptr);
   while((id = do_menu(comm_mn_ptr, M_GET_EVENT)) != DONE)
      {
      switch(id)
         {
         case PORT:
                  set_port(serial_ptr);
                  break;
         case SPEED:
                  set_speed(serial_ptr);
                  break;
         case PARITY:
                  set_parity(serial_ptr);
                  break;
         case BITS:
                  set_bits(serial_ptr);
                  break;
         case STOP_BIT:
                  set_stop(serial_ptr);
                  break;
         }
      m_show();
      }
   mn_close(comm_mn_ptr);
}
```

```
/*****************************************************************************/
/* This routine initializes and processes responses for the menu to send     */
/* command packets to the experiment hardware.                               */
/*****************************************************************************/

void mesg_menu(void)
{
   extern struct param serial;
   int id, psize, i, numpkts, result;
   MENU mesg_mn, *mesg_mn_ptr = &mesg_mn;
   unsigned char tpacket[MAX_PACKET];
   unsigned char rpacket[MAX_PACKET];
   unsigned char *comd[] = { "",
                             "\x30",             /* Initialize */
                             "\x10",             /* Data Dump  */
                             "\x70",             /* Status/ Health */
                             "\x90",             /* Testmode */
                             "\x20",             /* Endtestmode */
                             "\x50",             /* Shutdown */
                             "\x40\x00\x00\x00", /* Restart */
                             "\xA0",             /* Utime */
                             "\xF0\x00\x00"};    /* Parameter */

    menu_create(1, 6, 21, 17, M_VERTICAL, BACK_ATT, BDR_ATT,
                CSR_ATT, FIRST_ATT, SGL_BDR, WN_NORMAL, mesg_mn_ptr);
    item_add(" Initialize         ", INIT, 1, mesg_mn_ptr);
    item_add(" Data Dump          ", DUMP, 1, mesg_mn_ptr);
    item_add(" Status/Health      ", SOH, 8, mesg_mn_ptr);
    item_add(" Test Mode          ", TESTMODE, 1, mesg_mn_ptr);
    item_add(" End Test           ", ENDTESTMODE, 1, mesg_mn_ptr);
    item_add(" Shutdown           ", SDOWN, 1, mesg_mn_ptr);
    item_add(" Restart            ", RESTART, 1, mesg_mn_ptr);
    item_add(" Universal Time     ", UTIME, 1, mesg_mn_ptr);
    item_add(" Parameter          ", PARAMETER, 1, mesg_mn_ptr);
    item_add(" Exit               ", DONE, 2, mesg_mn_ptr);

    menu_set(mesg_mn_ptr);
    add_send_wn();
    add_rcv_wn();

    while((id = do_menu(mesg_mn_ptr, M_GET_EVENT)) != DONE)
       {
       if ( id == PARAMETER)
          get_param(comd[id]);  /* get params and stuff into packet */
       if (id == RESTART)
          get_days(comd[id]);

       if (id == PARAMETER)
        psize = 3;
       else psize = (id == RESTART ) ? 4 : 1;
       if(send_packet(tpacket, comd[id], psize,id) == TIMEOUT)
          {
          wait_prompt("Time Out!!  While sending packet...");
          continue;
          }
       if(serial.logstat) fprintf(lfp, "Sending....\n\n");
          disp_packet(send_ptr, tpacket, psize);
       if (validate_checksum(tpacket, psize + OVERHEAD))
          {
          wn_st("Checksum OK\n\n", send_ptr);
          if(serial.logstat) fprintf(lfp, "Checksum OK\n\n");
          }
       else
          {
          wn_st("Checksum Failed!\n\n", send_ptr);
          if(serial.logstat) fprintf(lfp, "Checksum Failed!\n\n");
```

```
          }
       if(serial.logstat) fprintf(lfp, "Received....\n\n");

       numpkts = (id == SDOWN) ? 2 : 1;

       for(i=0; i < numpkts; i++)
          {
          if ((result = rcv_packet(rpacket, &psize)) != OK)
             {
             disp_error_packet(rcv_ptr, rpacket, result);
             continue;
             }
          disp_packet(rcv_ptr, rpacket, psize);
          if (validate_checksum(rpacket, psize + OVERHEAD))
             {
             wn_st("Checksum OK\n\n", rcv_ptr);
             if(serial.logstat) fprintf(lfp, "Checksum OK\n\n");
             }
          else
             {
             wn_st("Checksum Failed!\n\n", rcv_ptr);
             if(serial.logstat) fprintf(lfp, "Checksum Failed!\n\n");
             }
          }
       }
    mn_close(mesg_mn_ptr);
    del_wn(send_ptr);
    del_wn(rcv_ptr);
}

/*****************************************************************************/
/* This routine initializes the menu and processes the responses to select   */
/* the communication port.                                                   */
/*****************************************************************************/

void set_port(struct param *s)
{
   int id;
   MENU setmenu;

   menu_create(42, 10, 50, 13, M_VERTICAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, SGL_BDR, WN_POPUP, &setmenu);
   item_add(" COM1 ", COM1, 4, &setmenu);
   item_add(" COM2 ", COM2, 4, &setmenu);
   menu_set(&setmenu);
   put_prompt("Select communication port.");
   id = do_menu(&setmenu, M_GET_EVENT);
   if(id) s->port = id;
   mn_close(&setmenu);
   del_wn(prompt_ptr);
}
```

```
/*****************************************************************************/
/* This routine initializes the menu and processes the responses to select   */
/* baud rate for the communication port.                                     */
/*****************************************************************************/

void set_speed(struct param *s)
{
   int id;
   MENU setmenu;

   menu_create(42, 11, 54, 15, M_VERTICAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, SGL_BDR, WN_POPUP, &setmenu);
   item_add(" 1200 Baud ", 12, 1, &setmenu);
   item_add(" 2400 Baud ", 24, 1, &setmenu);
   item_add(" 9600 Baud ", 96, 1, &setmenu);
   menu_set(&setmenu);
   id = do_menu(&setmenu, M_GET_EVENT);
   if(id) s->speed = id*100;
   mn_close(&setmenu);
}

/*****************************************************************************/
/* This routine initializes the menu and processes the responses to select   */
/* parity for the communication port.                                        */
/*****************************************************************************/

void set_parity(struct param *s)
{
   int id;
   MENU setmenu;

   menu_create(42, 12, 56, 16, M_VERTICAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, SGL_BDR, WN_POPUP, &setmenu);
   item_add(" No   parity ", NO_PARITY + 1, 1, &setmenu);
   item_add(" Even parity ", EVEN_PARITY + 1, 1, &setmenu);
   item_add(" Odd  parity ", ODD_PARITY + 1, 1, &setmenu);
   menu_set(&setmenu);
   id = do_menu(&setmenu, M_GET_EVENT);
   if(id) s->parity = id - 1;
   mn_close(&setmenu);
}

/*****************************************************************************/
/* This routine initializes the menu and processes the responses to select   */
/* the number of data bits for the communication port.                       */
/*****************************************************************************/

void set_bits(struct param *s)
{
   int id;
   MENU setmenu;

   menu_create(42, 13, 56, 18, M_VERTICAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, SGL_BDR, WN_POPUP, &setmenu);
   item_add(" 5 data bits ", 5, 1, &setmenu);
   item_add(" 6 data bits ", 6, 1, &setmenu);
   item_add(" 7 data bits ", 7, 1, &setmenu);
   item_add(" 8 data bits ", 8, 1, &setmenu);
   menu_set(&setmenu);
   id = do_menu(&setmenu, M_GET_EVENT);
   if (id) s->bits = id;
   mn_close(&setmenu);
}
```

```
/*****************************************************************************/
/* This routine initializes the menu and processes the responses to select   */
/* number of stop bits for the communication port.                           */
/*****************************************************************************/

void set_stop(struct param *s)
{
   int id;
   MENU setmenu;

   menu_create(42, 14, 56, 17, M_VERTICAL, BACK_ATT, BDR_ATT,
               CSR_ATT, FIRST_ATT, SGL_BDR, WN_POPUP, &setmenu);
   item_add(" 1 Stop bit  ", 1, 1, &setmenu);
   item_add(" 2 Stop bits ", 2, 1, &setmenu);
   menu_set(&setmenu);
   id = do_menu(&setmenu, M_GET_EVENT);
   if(id) s->stopbit = id;
   mn_close(&setmenu);
}

/*****************************************************************************/
/* This routine hides the menu specified by the parameter, restores the      */
/* screen beneath the menu, and releases the memory used by the menu.        */
/*****************************************************************************/

void mn_close(MENU *mn_ptr)
{
   m_hide();
   menu_restore(mn_ptr);
   menu_destroy(mn_ptr);
}

/*****************************************************************************/
/* This routine displays the communication port status in the status         */
/* window at the bottom of the screen.                                       */
/*****************************************************************************/

void disp_status(struct param *s)
{
   char parity_char;
   char *statstr[2] = {" PORT OFF", " PORT ON "};
   char *logstr[2] = {"  LOG OFF", "  LOG ON "};

   switch (s->parity)
      {
      case NO_PARITY: parity_char = 'N';
                  break;
      case EVEN_PARITY: parity_char = 'E';
                  break;
      case ODD_PARITY: parity_char = 'O';
                  break;
      }
   mv_cs(10, 0, status_ptr);
   wn_printf(status_ptr, "COM%d, %d, %c, %d, %d %s  %s Logfile:%s", s->port, s->s
peed,
          parity_char, s->bits, s->stopbit, statstr[s->status],
          logstr[s->logstat],s->logname);
}
```

```
/******************************************************************************/
/* This routine computes the checksum for the packet specified by the        */
/* parameter.  The result is stored in the checksum field of the packet.     */
/******************************************************************************/

void compute_checksum(unsigned char *packet)
{
   int temp, len, i;
   unsigned char c0 = 0, c1 = 0;

   temp = packet[PLEN];
   len =  (temp << 8) + packet[PLEN + 1] + HEAD_LEN;
   packet[len] = 0;
   packet[len + 1] = 0;
   for(i = 0; i < len + 2; i++)
      {
      c0 += packet[i];
      c1 += c0;
      }
   packet[len] = c0 - c1;
   packet[len + 1] = c1 - (c0 << 1);
}

/******************************************************************************/
/* This routine validates the checksum for the packet specified by the       */
/* parameter.  If the checksum is valid a TRUE value is returned.            */
/******************************************************************************/

int validate_checksum(unsigned char *packet, int len)
{
   int i;
   unsigned char c0 = 0, c1 = 0;

   for(i = 0; i < len; i++)
      {
      c0 += packet[i];
      c1 += c0;
      }
   return((c0 == 0) && (c1 == 0));
}

/******************************************************************************/
/* This routine gets the communication parameters that are stored in the     */
/* configuration file and stores them in the structure pointed to by the     */
/* parameter.                                                                 */
/******************************************************************************/

int get_config(struct param *s)
{
   FILE *fp;

   if ((fp = fopen("FERRO.CFG", "r")) == NULL)
      save_config(s);
   else
      {
      fscanf(fp, "%x\n%d\n%x\n%x\n%x\n%s\n%x", &(s->port), &(s->speed),
               &(s->parity), &(s->bits), &(s->stopbit), &(s->logname),
               &(s->logstat));
      fclose(fp);
      }
   /* if default config has log file turn on, open the logfile */
   if(s->logstat)
      fopen(s->logname, "at+");
   return(0);
}
```

```
/******************************************************************************/
/* This routine saves the current communication parameters to the           */
/* configuration file.                                                        */
/******************************************************************************/

int save_config(struct param *s)
{
   FILE *fp;

   if ((fp = fopen("FERRO.CFG", "w")) == NULL)
      wait_prompt("Unable to open configuration file.");
   else
      {
      fprintf(fp, "%x\n%d\n%x\n%x\n%x\n%s%x", s->port, s->speed, s->parity,
                  s->bits, s->stopbit, s->logname, s->logstat);
      fclose(fp);
      }
   return(0);
}

/******************************************************************************/
/* This routine creates a generic window to be used for prompts.             */
/******************************************************************************/

void put_prompt(char *prompt)
{
   wn_create( 1, 20, 78, 22, DBL_BDR, WN_NORMAL, prompt_ptr);
   wn_color(YELLOW, BLUE, prompt_ptr);
   wn_bdr_color(WHITE, BLUE, prompt_ptr);
   add_window(prompt_ptr);
   wn_plst(CENTERED, 0, prompt, prompt_ptr);
}

/******************************************************************************/
/* This routine creates a prompt window, displays a message, and waits for   */
/* the user to respond by pressing any key.                                   */
/******************************************************************************/

void wait_prompt(char *prompt)
{
   wn_create( 1, 20, 78, 23, DBL_BDR, WN_NORMAL, prompt_ptr);
   wn_color(YELLOW, BLUE, prompt_ptr);
   wn_bdr_color(WHITE, BLUE, prompt_ptr);
   add_window(prompt_ptr);
   wn_plst(CENTERED, 0, prompt, prompt_ptr);
   wn_plst(CENTERED, 1, "Press any key to continue", prompt_ptr);
   wait_event();
   del_wn(prompt_ptr);
}
```

```
/****************************************************************************/
/* This routine initializes and displays the transmit window.               */
/****************************************************************************/

void add_send_wn(void)
{
   wn_create( 22, 2, 78, 12, DBL_BDR, WN_NORMAL, send_ptr);
   wn_color(YELLOW, BLUE, send_ptr);
   wn_bdr_color(WHITE, BLUE, send_ptr);
   wn_name(" Send ", send_ptr);
   add_window(send_ptr);
}

/****************************************************************************/
/* This routine initializes and displays the receive window.                */
/****************************************************************************/

void add_rcv_wn(void)
{
   wn_create( 22, 13, 78, 23, DBL_BDR, WN_NORMAL, rcv_ptr);
   wn_color(YELLOW, BLUE, rcv_ptr);
   wn_bdr_color(WHITE, BLUE, rcv_ptr);
   wn_name(" Receive ", rcv_ptr);
   rcv_ptr->eol_wrap = 1;
   add_window(rcv_ptr);
}

/****************************************************************************/
/* This routine clears and releases the memory used by the window specified */
/* by the parameter.                                                        */
/****************************************************************************/

void del_wn(WINDOW *wnptr)
{
   wn_clear(wnptr);
   remove_window(wnptr);
   wn_destroy(wnptr);
}

/****************************************************************************/
/* This routine displays a prompt window and gets the digits for Cycle #    */
/* and CycleDay to send in the Restart command packet.                      */
/****************************************************************************/

int get_days(unsigned char *days)
{
   unsigned char cyclestr[43] = "Cycle: 00   Day in cycle: 00";
   int c,d, result;

   wn_create( 25, 10, 83, 15, DBL_BDR, WN_NORMAL, temp_ptr);
   wn_color(YELLOW, BLUE, temp_ptr);
   wn_bdr_color(WHITE, BLUE, temp_ptr);
   add_window(temp_ptr);
   wn_plst(CENTERED, 0, "Enter Cycle and Cycle Day:", temp_ptr);
   wn_csr_pos(3,5,temp_ptr);
   result = wn_gets(cyclestr, "Cycle:      Day in cycle: __",
          "       ##                 ##",  temp_ptr->att, STRIP_ON, temp_ptr);
   sscanf(cyclestr,"%2d%2d%2d",&c,&d,&e);
   days[1] = (unsigned char) c;
   days[2] = (unsigned char) d;
   del_wn(temp_ptr);
   return(result);
}

/****************************************************************************/
/* This routine displays a prompt window and gets the digits to send in the */
```

```
/* parameter packet.                                                        */
/****************************************************************************/

void get_param(unsigned char *params)  /* add to term.c */
   {

      unsigned char paramstr[25] = "Byte1: 00 Byte2: 00";
      int c,d;

      wn_create(25,10,65,15, DBL_BDR, WN_NORMAL, temp_ptr);
      wn_color(YELLOW, BLUE, temp_ptr);
      wn_bdr_color(WHITE,BLUE,temp_ptr);
      add_window(temp_ptr);
      wn_plst(CENTERED, 0, "ENTER PARAMETER BYTES:", temp_ptr);
      wn_csr_pos(3,5, temp_ptr);
      wn_gets(paramstr, "Byte1: __ Byte2: __ ",
                      "        UU        UU", temp_ptr->att,STRIP_ON, temp_ptr);
      sscanf(paramstr, "%2x%2x",&c,&d);
      params[1] = c;  /* stuff pkt w/ params */
      params[2] = d;
      del_wn(temp_ptr);
   }

/****************************************************************************/
/* This routine displays a prompt window to get a new name for the log      */
/* file.  No file extension is should be entered.                           */
/****************************************************************************/

void get_logname(struct param *s)
{
   char logstr[25] = "File name:  ________.log";

   wn_create( 15, 10, 70, 15, DBL_BDR, WN_NORMAL, temp_ptr);
   wn_color(YELLOW, BLUE, temp_ptr);
   wn_bdr_color(WHITE, BLUE, temp_ptr);
   add_window(temp_ptr);
   wn_plst(CENTERED, 0, "Enter the new name of the LOG File.", temp_ptr);
   wn_plst(CENTERED, 1, "Use NO filename extention (defaults to .log).",
                                                                  temp_ptr);
   wn_csr_pos(3,5,temp_ptr);
   wn_gets(logstr, "File name:  ________.log",
    "           UUUUUUUU    ", temp_ptr->att, STRIP_ON, temp_ptr);
   sscanf(logstr,"%8s",s->logname);
   strcat(s->logname, ".log");
   del_wn(temp_ptr);
}
```

```
#include "term.h"
/*****************************************************************************/
/* This routine gets a character from the serial port.  If a response is    */
/* not received within the time-out period, an error message is displayed   */
/* and an error code is returned.                                            */
/*****************************************************************************/

int getc_serial(unsigned char *ch)
{
   time_t btime;
   int result;

   btime = time(NULL);
   while((result = getc_buf()) == ERROR)
      if ((time(NULL) - btime) > TIME_OUT)
         {
            wait_prompt("TIME OUT!  Waiting for char from serial port.");
         return(ERROR);
         }
   *ch = (unsigned char) result;
   return(0);
}

/*****************************************************************************/
/* This routine assembles and transmits a packet.  Pointers to the packet   */
/* buffer and the data to be transmitted are passed as parameters as is the */
/* length of the data field.                                                 */
/*****************************************************************************/

int send_packet(unsigned char *pkt, unsigned char *data, unsigned char data_len,u
nsigned int id)
{
   int i;

   pkt[0] = FLAG;
   pkt[PLEN] = (unsigned char)((data_len & 0xff00) >> 8 );
   pkt[PLEN + 1] = (unsigned char) (data_len & 0x00ff);
    pkt[3] = 0x4d;               /* hi byte ferro address */
    pkt[4] = 0x01;               /* low byte ferro address */
    pkt[5] = 0x6d;               /* hard code source address */
    pkt[6] = 0x01;               /* low byte source address */
    pkt[7] = (id == SOOWN | id == DUMP)? 0x00 : 0x08; /* set cntrl field */
                                                      /* send no ack - 00, se
nd ack 08 */
    pkt[8] = 0x00;    /* not used  */
    for(i=0; i < data_len ; i++)
         pkt[i + HEAD_LEN] = data[i];
    compute_checksum(pkt);
    if(serial_out(pkt[0]) == TIME_OUT)              /* transmit FLAG */
         return(TIME_OUT);
    for(i=1; i < OVERHEAD + data_len; i++)              /* xmit rest of packet */

         {
         if(serial_out(pkt[i]) == TIME_OUT)
             return(TIME_OUT);
         else
             if(pkt[i] == FLAG)                      /* xmit double flag if */
                  if(serial_out(FLAG) == TIME_OUT)     /* not beginning of pckt
 */
               return(TIME_OUT);
      }
   return(OK);
}
```

```
/*****************************************************************************/
/* This routine receives a packet from the experiment board, checking for   */
/* errors along the way.                                                     */
/*****************************************************************************/

int rcv_packet(unsigned char *packet, int *data_len)
{
   time_t btime;
   int i, idx = 0, result;
   int state;
   unsigned char ch;

   for(i=0; i< MAX_PACKET; packet[i++] = '\x0');
   btime = time(NULL);
   /* wait for beginning of packet flag */
   while(((result = getc_serial(&ch)) != ERROR) && ch != FLAG)
      if ((time(NULL) - btime) > TIME_OUT)
         {
         /* Timed out waiting for FLAG */
         wait_prompt("TIME OUT!  Did not receive beginning of packet flag.");
         return(idx);
         }
   if(result == ERROR) return(idx);            /* Didn't receive char */
   packet[idx++] = ch;


/* compute size of data field */
   if((result = get_packet_byte(&ch)) == ERROR) return(idx);
   *data_len = ((unsigned int) ch << 8);
   packet[idx++] = ch;

   if((result = get_packet_byte(&ch)) == ERROR) return(idx);
   *data_len += (unsigned int) ch;
   packet[idx++] = ch;

   while(idx < *data_len + OVERHEAD)
      {
      if((result = get_packet_byte(&ch)) == ERROR) return(idx);
      if(result == FLAG)
         { packet[idx++] = FLAG;   /* necessary to disp all chars rec'd */
           packet[idx++] = ch;     /* on a single FLAG error            */
              return(idx);
      }
      packet[idx++] = ch;
      }
   return(OK);
}
```

```
/*************************************************************************/
/* This routine gets a byte from the serial port using the getc_serial     */
/* function.  If a FLAG character is received, a second FLAG is checked    */
/* for.  If the second FLAG is received, the character is returned.        */
/* Otherwise, an error message is displayed and an error code returned.    */
/*************************************************************************/

int get_packet_byte(unsigned char *ch_ptr)
{

   if(getc_serial(ch_ptr) == ERROR) return(ERROR);
   if(*ch_ptr == FLAG)
      {
      if(getc_serial(ch_ptr) == ERROR) return(ERROR);
      if(*ch_ptr != FLAG)
         {
         wait_prompt("ERROR! Received single FLAG char in body of packet");
         return(FLAG);
         }
      }
   return(OK);
}

/*************************************************************************/
/* This routine displays a packet that has been received.  The packet is  */
/* parsed and labels are displayed in the window to highlight the header, */
/* data, and checksum fields.  If the logging has been selected, all      */
/* displayed characters are echoed to the log file.                       */
/*************************************************************************/

void disp_packet(WINDOW *wn_ptr, unsigned char *pk, unsigned int data_len)
{
   int i = 0;
   extern struct param serial;
   extern FILE *lfp;
   wn_printf(wn_ptr, "Header: ");
   if(serial.logstat) fprintf(lfp,"Header: ");

   while (i < HEAD_LEN)
      {
      wn_printf(wn_ptr, "%02X ", pk[i]);
      if(serial.logstat) fprintf(lfp, "%02X ", pk[i]);
      i++;
      }
   while (i < data_len + HEAD_LEN)
      {
      if(!((i-HEAD_LEN) % 10))
         {
         wn_st_qty("\nData: ", 10, wn_ptr);
         if(serial.logstat) fprintf(lfp, "\nData: ");
         }
      wn_printf(wn_ptr,"%02X ", pk[i]);
      if(serial.logstat) fprintf(lfp, "%02X ", pk[i]);
      i++;
      }
   wn_printf(wn_ptr, "\nChecksum: %02X %02X           ", pk[i], pk[i+1]);
   if(serial.logstat) fprintf(lfp, "\nChecksum: %02X %02X           "
                                                          pk[i], pk[i+1]);
}
```

```
/*************************************************************************/
/* If an error is detected during the reception of a packet, this routine  */
/* is called to display all bytes that have been received to the point of  */
/* error, including the byte that caused the error.                        */
/*************************************************************************/

void disp_error_packet(WINDOW *wn_ptr, unsigned char *pk, int data_len)
{
   int i;
   extern struct param serial;
   extern FILE *lfp;

   wn_st("Error during packet reception\n", wn_ptr);
   wn_st("The following bytes were received:\n", wn_ptr);
   if(serial.logstat)
      {
      fprintf(lfp, "Error during packet reception\n");
      fprintf(lfp, "Following bytes rec'd before error:\n");
      }
   if(data_len >= 0)
      for(i=0; i < data_len; i++)
         {
         if(!(i % 10))
            {
            wn_st_qty("\nData: ", 10, wn_ptr);
            if(serial.logstat) fprintf(lfp, "\nData: ");
            }
         wn_printf(wn_ptr,"%02X ", pk[i]);
         if(serial.logstat) fprintf(lfp, "%02X ", pk[i]);
         }
}
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
Cameron Station
Alexandria, Virginia 22304-6145

2. Library, Code 52 2
Naval Postgraduate School
Monterey, California 93943-5000

3. Chairman, Code EC 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, Calfornia 93943-5000

4. Chairman, Code SP 1
Space Systems Academic Group
Naval Postgraduate School
Monterey, Calfornia 93943-5000

5. Prof. F. Terman, Code EC/Tz 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, Calfornia 93943-5000

6. LT Terry G. Tutt 1
107 Acorn Court
Kingsland, Georgia 31548

845-216

Thesis
T95834 Tutt
c.1 Design of microcontroller software for a satellite-based ferroelectric capacitor experiment.

Thesis
T95834 Tutt
c.1 Design of microcontroller software for a satellite-based ferroelectric capacitor experiment.

AMERICAN BINDERY
1
10-93

DUDLEY KNOX LIBRARY

3 2768 00033181 3